NAVAL POSTGRADUATE SCHOOL
Monterey, California

THESIS

NPSNET: Real-Time 3D Ground-Based Vehicle Dynamics

by

Hyun Kyoo Park
March 1992

Thesis Advisor: Dr. Michael J. Zyda
Thesis Co-Advisor: David R. Pratt

CURITY CLASSIFICATION OF THIS PAGE

REPORT DOCUMENTATION PAGE

a. REPORT SECURITY CLASSIFICATION: UNCLASSIFIED	1b. RESTRICTIVE MARKINGS
2a. SECURITY CLASSIFICATION AUTHORITY	3. DISTRIBUTION/AVAILABILITY OF REPORT: Approved for public release; distribution is unlimited
2b. DECLASSIFICATION/DOWNGRADING SCHEDULE	
4. PERFORMING ORGANIZATION REPORT NUMBER(S)	5. MONITORING ORGANIZATION REPORT NUMBER(S)
6a. NAME OF PERFORMING ORGANIZATION: Dept. of Computer Science, Naval Postgraduate School; 6b. OFFICE SYMBOL (if applicable): CS	7a. NAME OF MONITORING ORGANIZATION: Naval Postgraduate School
6c. ADDRESS (City, State, and ZIP Code): Monterey, CA 93943-5000	7b. ADDRESS (City, State, and ZIP Code): Monterey, CA 93943-5000
8a. NAME OF FUNDING/SPONSORING ORGANIZATION; 8b. OFFICE SYMBOL (if applicable)	9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER
8c. ADDRESS (City, State, and ZIP Code)	10. SOURCE OF FUNDING NUMBERS: PROGRAM ELEMENT NO. / PROJECT NO. / TASK NO. / WORK UNIT ACCESSION N

11. TITLE (Include Security Classification)
NPSNET: REAL-TIME 3D GROUND-BASED VEHICLE DYNAMICS

12. PERSONAL AUTHOR(S)
Park, Hyun Kyoo, Captain Korean Army

13a. TYPE OF REPORT	13b. TIME COVERED	14. DATE OF REPORT (Year, Month, Day)	15. PAGE COUNT
Master's Thesis	FROM 03/90 TO 03/92	1992 March 26	74

16. SUPPLEMENTARY NOTATION
The views expressed in this thesis are those of the authors and do not reflect the official policy or position of the Department of Defense or the United States Governme

17. COSATI CODES			18. SUBJECT TERMS (Continue on reverse if necessary and identify by block number)
FIELD	GROUP	SUB-GROUP	Graphics, Simulation, Motion Control, Dynamics, Behavioral Motion

19. ABSTRACT (Continue on reverse if necessary and identify by block number)

The Naval Postgraduate School (NPS) has been developing real-time 3D visual simulators on inexpensive comme cially available graphics workstations. The effort to develop a 3D visual simulation system, NPSNET, is an explora tion and experimentation virtual worlds. Virtual world system have many goals including military training. This wor is part of NPSNET. The current NPSNET system is kinematically based. The objectives of this work are motion dy namics and behavioral motion control for autonomous vehicles. Motion control is difficult when using dynamics du to the internal and external forces, however it enhances the realism as well as motion accuracy. We develop motio dynamics and behavioral control for ground-based vehicles. Motion dynamics and behavioral motion are an essenti part of NPSNET for realistic battlefield simulation.

20. DISTRIBUTION/AVAILABILITY OF ABSTRACT	21. ABSTRACT SECURITY CLASSIFICATION	
[X] UNCLASSIFIED/UNLIMITED [ ] SAME AS RPT. [ ] DTIC USERS	UNCLASSIFIED	
22a. NAME OF RESPONSIBLE INDIVIDUAL: Dr. Michael J. Zyda	22b. TELEPHONE (Include Area Code): (408) 646-2305	22c. OFFICE SYMBOL: CS/Zk

DD FORM 1473, 84 MAR — 83 APR edition may be used until exhausted. All other editions are obsolete.

SECURITY CLASSIFICATION OF THIS PAGE
UNCLASSIFIED

Approved for public release; distribution is unlimited

NPSNET: Real-Time 3D Ground-Based Vehicle Dynamics

by

Hyun Kyoo Park
Captain, Korean Army
B.S., Korea Military Academy, 1987

Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF COMPUTER SCIENCE

from the

NAVAL POSTGRADUATE SCHOOL
March 1992

ABSTRACT

The Naval Postgraduate School (NPS) has been developing real-time 3D visual simulators on inexpensive commercially available graphics workstations. The effort to develop a 3D visual simulation system, NPSNET, is an exploration and experimentation virtual worlds. Virtual world system have many goals including military training. This work is part of NPSNET. The current NPSNET system is kinematically based. The objectives of this work are motion dynamics and behavioral motion control for autonomous vehicles. Motion control is difficult when using dynamics due to the internal and external forces, however it enhances the realism as well as motion accuracy. We develop motion dynamics and behavioral control for ground-based vehicles. Motion dynamics and behavioral motion are an essential part of NPSNET for realistic battlefield simulation.

TABLE OF CONTENTS

LIST OF FIGURES

ACKNOWLEDGEMENTS

I would like to thank Korean government and Korean Army. I have come to the Naval Postgraduate School to study computer science for two years with their grant. Also, my sincere thanks to the U.S Department of Navy. I always thank my loving family for their encouragement from the beginning.

This work would not have been done without useful help from many people. I would like to thank Dr. Michael Zyda for his guidance from the first to the end of this work and Mr. David Pratt for his aid during the entire implementation. Lastly, I also thank Dr. Sehung Kwak for useful discussions concerning related work and the motivation of this research.

I. INTRODUCTION

A. MOTIVATION FOR RESEARCH

Recently simulation technology has been adapted in many ways. The flight simulator is used for training pilots and astronauts to safely control complex, expensive vehicles through simulated environments [Deyo, 89]. The capability of graphics workstations drops the relative cost of simulation, hence the technology has been moving gradually toward lower cost simulation environments through which the operator is also able to control his own viewpoint or motion.

An advanced ground vehicle simulator is made up of a few, expensive subsystems that combine to create a complete virtual environment for the operator. Interactive real-time ground vehicle simulation demands scene realism and accuracy of motion. Because of the rapid development of computer technology, current computer hardware and software make it possible to synthesize realistic images. However, the real-time physical simulation of motion still remains a difficult task. It is now possible to use vehicle dynamics for simulating real world motion on a general purpose workstation.

Dynamics can produce complex motion with minimal user inputs, but it is problematic in the difficulty of motion control and it is computationally expensive. A more automatic method of controlling motion can improve the control problem. Behavioral simulation is a means of automatic motion control according to certain rules. We can use behavioral control for autonomous moving vehicles in the simulation system. The research of motion dynamics and autonomous motion planning of a vehicle simulator on a general purpose workstation is a key component for the construction of a fully interactive virtual world.

B. NPSNET

The Defense Advanced Research Projects Agency (DARPA) has been developing a simulation system entitled Simulation Networking (SIMNET) [Thorpe, 87]. SIMNET is an interactive battlefield simulation system designed for military training. SIMNET nodes require visual displays and specific hardware of a particular vehicle. The effort to develop a low-cost SIMNET type system based on commercially available graphic workstations has been an ongoing project at the Naval Postgraduate School (NPS). The system, NPSNET, is a real-time, 3D visual simulation system capable of displaying vehicle movement over the ground or in the air. NPSNET runs on Silicon Graphics workstations attached to a local area Ethernet and uses SIMNET databases and networking formats [Zyda, et al., 92].

In NPSNET, up to 500 of the vehicles can be driving in the world at one time. The user can select any one of the active vehicles via mouse selection and control it with a six degree of freedom SpaceBall or button and dialbox. The pick button on the SpaceBall fires a round from the driven vehicle.Vehicles can be controlled by a prewritten script, driven interactively from other workstations, or autonomously via computer control. Displays show on-ground cultural features such as roads, buildings, and soil types. These vehicles move around in the virtual environment that is based on the terrain at Fort Hunter-Liggett, California.

Depending upon the model of IRIS being used, the user can select to use texturing, and environmental effects such as fog and haze. A two dimensional (2D) map can be displayed that shows the position and tracking of all the players. This map displays the direction and viewing triangle of the driven vehicle as well as the position and movement of the remaining vehicles. The statistics and data concerning the driven vehicle (speed, pitch, roll, and so on) are displayed in an information window at the top of the screen.

C. OBJECTIVES OF THIS WORK

The current NPSNET system is kinematically-based. The motion description, from user-input through actual frame rendering, consists of positions specified over time. The goal of this work is to add motion dynamics to NPSNET. The aerodynamics model is being developed concurrently for air vehicles [Cooke, 92]. This work concentrates on the essential system issues for the implementation of virtual environments for real-time ground based vehicle simulation.

1. Dynamics

Dynamics deals with the motion of bodies under the action of forces. Motion occurs in the physical world as result of forces acting on objects. To achieve a degree of realism in motion control, the motion of objects must be simulated by the physical principles of dynamics. This work focuses on theoretical and numerical aspects of the implementation of a dynamic motion simulation system.

2. Behavioral Control

The search for more automatic methods of controlling motion is a major area in computer graphics. The problem of motion planning of autonomous vehicles consists of selecting the geometric path and vehicle speeds so as to avoid obstacles and it should minimize computationally expensive functions to keep the frame rate at an acceptable level. Motion is planned at a task-level and computed using physical laws. Inverse dynamics is the essence of the autonomous motion control. In higher level motion control, behavioral motion in particular, evaluates the state of the environment and generates motion according to certain rules of behavior [Wilhelms, 90].

3. Vehicle Parameterization

The vehicles used in NPSNET are military vehicles such as tanks, APCs and trucks. To get the realism and satisfy the simulation system users, the vehicles' motion

should match the real vehicle specification. These specifications are defined in the data structure of the vehicles and incorporated with the dynamics equations.

D. THESIS ORGANIZATION

Chapter II covers basic dynamics in the vehicle simulation. Derived dynamics equations are defined and an integration method is presented. Chapter III provides vehicle modeling and motion control issues including dynamics as well as vehicle constraints. Chapter IV presents the high-level motion control issues, especially behavioral motion with inverse dynamics. Chapter V provides implementation details of the dynamic motion control and data structures into the simulator. Chapter VI covers conclusions and suggestions for further work in this area.

II. DYNAMICS

A. INTRODUCTION

This chapter presents the rigid body dynamics and coordinate system that are used in the simulation. A rigid body can be described as a fixed and unchanging extended mass. The dynamics of general plane motion of a rigid body combines translation and rotation. In the vehicle dynamics analysis, vehicles are treated as extended rigid bodies, and information concerning their mass, centers of mass, and mass distribution is needed. Previously, the dynamics equations of a vehicle model are derived [Meriam et.al., 86][1]. Motion is produced by a simplified simulation of dynamics, that describes the linear and angular accelerations of a rigid body in terms of the forces acting on them.

If a number of forces are acting on the body, their total translational effect can be found by merely summing them. The center of mass of the body will move translationally as if it were a particle mass influenced by one net force. The external forces acting on the vehicle consist of the frictional forces between the vehicle and ground, the normal forces, and the gravity force. A torque is similar to a force, except that it causes a rotational motion about a particular axis. Torques can be represented as 3D vectors describing their components about an x, y, and z-axis.

B. DYNAMICS OF RIGID BODY MOTION

The body coordinate system is used in the simulation. Since a rigid body is free to rotate and translate, it is possible to describe the placement of a rigid body in world space in terms of the location of the body's center of mass and the body's rotational orientation in world space.

---

[1] The mathematical dynamics equations we cover below are taken from this reference and then implemented with vehicle constraints.

The basis vectors of this coordinate system are usually the principal axis of the object. For body i, the center of mass is represented by the vector $c(t)$, and its orientation is represented by the 3×3 matrix R. The terms that are used in this chapter can be found in the Table 2.1.

I	Inertia Tensor Matrix
f	force
f_{grv}	Force due to gravity
f_{ext}	External applied force
τ	Torque
c	Center of Mass
r	Position Vector
R	Orientation Matrix
v	Linear velocity
a	Linear acceleration
p	Linear momentum
w	Angular velocity
$\dot{w}$	Angular acceleration
l	Angular momentum
m	mass
Δt	Time step between samples
l_x, l_y, l_z	Moments of inertia
l_{xy}, l_{yz}, l_{xz}	Products of inertia

Table 2.1. Dynamics Terms

1. Center of Mass

A representative particle of mass, m_i, is located by its position vector, r_i, of the reference axis. The center of mass of the system of particles is located by the position vector, r, which, from the definition of the mass center as covered in statics, is given by

$$m\ddot{r} = \Sigma m_i r_i \qquad \text{(Eq. 2.1)}$$

where the total mass $M = \Sigma m_i$ [Meriam et. al., 86].

At time *t*, the center of mass is location in world space given by the weighted sum

$$\frac{\Sigma m_i r_i(t)}{\Sigma m_i} = \frac{\Sigma m_i r_i(t)}{M} \qquad \text{(Eq. 2.2)}$$

If using the body coordinate system, the center of mass in body coordinate is at the origin. This means that the body space is

$$\frac{\Sigma m_i r_i(t)}{M} = \vec{O} \qquad \text{(Eq. 2.3)}$$

which means that $\Sigma m_i r_{0i} = \vec{O}$ as well.

In the motion simulation, the body coordinate system is used for the following relations. Let *c(t)* be the location of the center of mass in world space. Since the center of the mass is located at *c(t)*, the location of the i_{th} mass point in world space is $r_i(t) = R(t) r_{0i} + c(t)$. At time *t*, since $R(t)\sum m_i r_{0i} = R(t)\vec{O} = \vec{O}$, the Equation 2.2 is

$$\frac{\Sigma m_i r_i(t)}{M} = \frac{\Sigma m_i (R(t) r_{0i} + c(t))}{M} = c(t)\frac{\Sigma m_i}{M} = c(t) \qquad \text{(Eq. 2.4)}$$

$$\sum m_i (r_i(t) - c(t)) = \sum m_i (R(t) r_{0i} + c(t) - c(t)) = R(t)\sum m_i r_{0i} = \vec{O} \qquad \text{(Eq. 2.5)}$$

These relations enable us to consider linear translation separately from angular orientation. Thus it is possible to combine a rotation followed by a translation without changing the size or shape of the body. This can simplify the description of the laws of motion.

2. Linear Momentum

The linear momentum *p* of a particle with mass *m* and velocity *v* is defined as

$$p = mv \qquad \text{(Eq. 2.6)}$$

Since the derivative of velocity is acceleration *a*, Newton's law of motion for a particle mass can be expressed as

$$f = ma = m\frac{dv}{dt} = \frac{d}{dt}(mv) = \frac{dp}{dt} \qquad \text{(Eq. 2.7)}$$

This law says that the rate of change of the momentum p of a particle is equal to the force f acting on the particle. Since $f = p$ for a single particle, it is obvious to expect that a similar law holds for rigid bodies, comprised of collections of particles.

3. Angular Momentum

There should be a linear relation between the body's angular momentum and its rotational velocity. The angular momentum depends on the choice of origin of coordinates. The angular momentum $l_i(t)$ of the i_{th} particle of the rigid body is

$$l_i(t) = (r_i(t) - c(t)) \times m_i(r_i(t) - c(t)) \qquad \text{(Eq. 2.8)}$$

Thus, $l_i(t)$ is the cross product of the i_{th} particle's displacement from the center of mass and the momentum of the particle with respect to the center of mass. The total angular momentum $L(t)$ of the body is simply

$$L(t) = \sum l_i(t) = \sum (r_i(t) - c(t)) \times m_i(r_i(t) - c(t)) \qquad \text{(Eq. 2.9)}$$

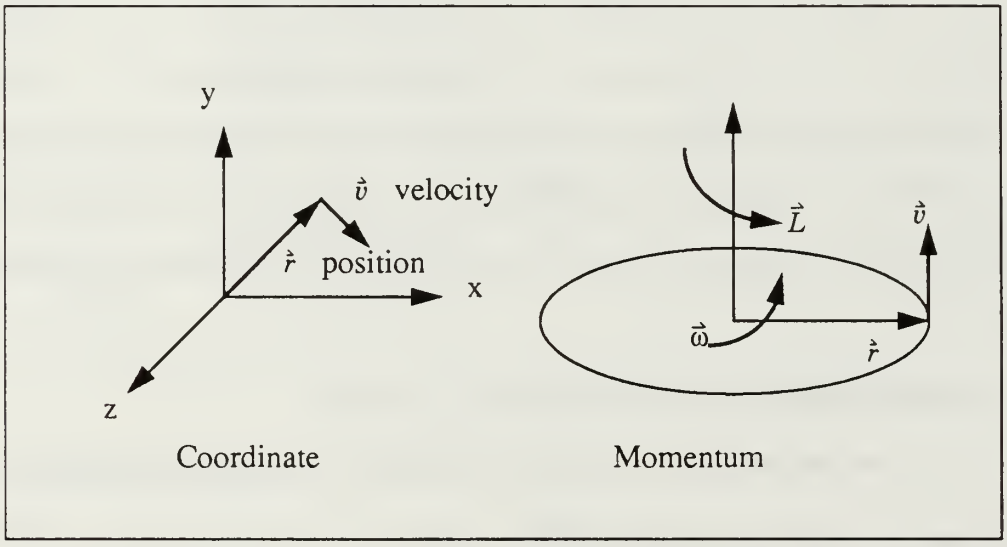


Figure 2.1 Cartesian Coordinate System And Momentum Of Particle

4. Inertia Tensor

The inertia tensor of a body is a 3×3 matrix that describes the distribution of mass in the body. For symmetrical bodies, there are simple ways of calculating the moments of inertia. For a box centered at the origin with width c in x, b in y and a in z, multiplying the density by the volume gives the mass. In this case, the moment of inertia around the origin of x-axis is

$$I_x = \frac{1}{12} m (a^2 + b^2) \qquad \text{(Eq. 2.10)}$$

Similarly for y and z-axis, $I_y = \frac{1}{12} m (a^2 + c^2)$ and $I_z = \frac{1}{12} m (b^2 + c^2)$. The off-diagonal terms, such as I_{xy}, are

$$I_{xy} = \int_{\frac{-x_0}{2}}^{\frac{x_0}{2}} \int_{\frac{-y_0}{2}}^{\frac{y_0}{2}} \int_{\frac{-z_0}{2}}^{\frac{z_0}{2}} \rho (x, y, z) (xy) dxdydz = \int_{\frac{-x_0}{2}}^{\frac{x_0}{2}} \int_{\frac{-y_0}{2}}^{\frac{y_0}{2}} \int_{\frac{-z_0}{2}}^{\frac{z_0}{2}} xydxdydz = 0 \qquad \text{(Eq. 2.11)}$$

and similarly for the others because the integrals are all symmetric. Thus for the symmetrical block with width a in x, b in y and c in z, the inertia tensor matrix is

$$I_{body} = \frac{M}{12} \begin{bmatrix} b^2 + c^2 & 0 & 0 \\ 0 & a^2 + c^2 & 0 \\ 0 & 0 & a^2 + b^2 \end{bmatrix} \qquad \text{(Eq. 2.12)}$$

[Baraff, 91].

C. NEWTON-EULER EQUATIONS

1. Linear and Angular Acceleration

Two parts of Newton-Euler equations are the translational motion of its center of mass and rotational motion about the center of mass. The linear momentum and angular momentum are used in the simulation for the linear velocity and angular orientation each.

By Newton's law, the linear accelerations for x and z axis of the center of mass are derived from Equation 2.7. If c is the vehicle's center of mass position and m is total mass, the equation of motion for linear acceleration a is second derivative of position c.

$$f_x = ma_x, \quad f_z = ma_z \qquad \text{(Eq. 2.13)}$$

where a is in meter/second^2.

The torque, τ, differs from a force in that the torque on a particle depends on the location of the particle. The torque acts on the particle, i, is $\tau = (r_i(t) - c(t)) \times F$. By the Equation 2.11, the products of inertia are zero. So, the rotational equations for motion about the center of mass are,

$$\tau_x = I_x\dot{\omega}_x + (I_z - I_y)\,\omega_y\omega_z \qquad \text{(Eq. 2.14)}$$

$$\tau_y = I_y\dot{\omega}_y + (I_x - I_z)\,\omega_x\omega_z \qquad \text{(Eq. 2.15)}$$

$$\tau_z = I_z\dot{\omega}_z + (I_y - I_x)\,\omega_x\omega_y \qquad \text{(Eq. 2.16)}$$

where the vectors are expressed in the principal axis frame of the vehicle. In the simulation, ω is in radians/second and $\dot{\omega}$ in radians/second^2 [Wilhelms, 87].

The dynamics equations are solved for new angular and linear accelerations. These accelerations must then be integrated to find new velocities and integrated again to find new positions.

2. Integration

The Euler method uses six equations shown in Equation 2.13, 2.14, 2.15, 2.16. Because of real-time limitations, methods that require only a single updated evaluation of f for each time step are currently used. The Euler method is used for the integration.

$$v_{t+\delta t} = v_t + a_t\delta t \qquad \text{(Eq. 2.17)}$$

$$\theta_{t+\delta t} = \theta_t + \omega_t\Delta t + 0.5\omega_t\Delta t^2 \qquad \text{(Eq. 2.18)}$$

$$c_{t+\delta t} = c_t + v_t\Delta t + 0.5a_t\Delta t^2 \qquad \text{(Eq. 2.19)}$$

where θ and c are the orientation and position at time t and $t + \Delta t$.

The Euler method has two advantages. It is simple to program and efficient. The velocity is calculated once in a time step. Also it is efficient. This assumes acceleration is constant over the time period. The disadvantage of the method is the assumption that the

velocity is varied slowly in a time step. The inaccuracy problem, discontinuity between the current approximation speed and the next approximation speed, occurs when the time period is large or accelerations are changing rapidly. However with reasonably small time steps, the Euler method can produce appropriate result without too much trouble arising [Wilhelms, 87].

In the simulation, the Euler method is used for this reason and the result is reasonable for ground based vehicles. The other method is investigated and implemented for air vehicles and both methods are compared in [Cooke, 92].

III. VEHICLE MODELING AND CONSTRAINTS

A. VEHICLE MODELING

In NPSNET, there are two types of ground vehicles: tracked and wheeled. All the vehicles can be approximated as rectangular blocks with the same mass density. The center of mass is assumed to be the center of the block. This assumption can reduce the dynamics calculation without the consideration of the vehicle types. The reference and path coordinates are shown at Figure 3.1.

Body coordinates in the simulation for the vehicle description are subject to holonomic constraints[2]. The direction of the wheeled vehicle is computed by its front wheel angle, however the tracked vehicle's direction is changed by the difference between the left and right track force. The motion of a vehicle is also limited by its constraints. In translational motion, the maximum acceleration is not always positive, nor is the maximum deceleration negative. This depends on the slope of the path. The engine forces of vehicles moving on bumpy terrain need to be carefully selected in order to maintain continuous contact with the ground as well as maintaining speed [Shiller et. al., 91].

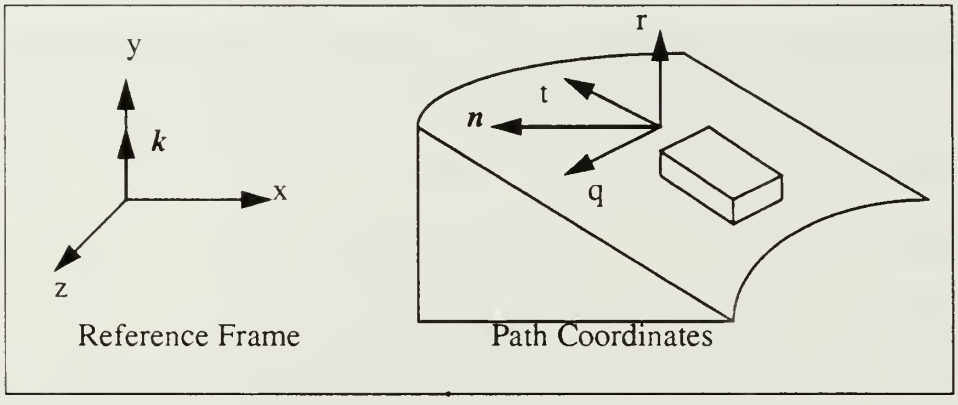


Figure 3.1 Reference Frame And Path Coordinates

[2] Holonomic constraints mean $\Phi(c_i) = 0$ (constraints in precisely this functional form which can be used to eliminate dependent coordinates. c is body coordinates). Vehicles are assumed to be block structures.

1. Tracked Vehicle

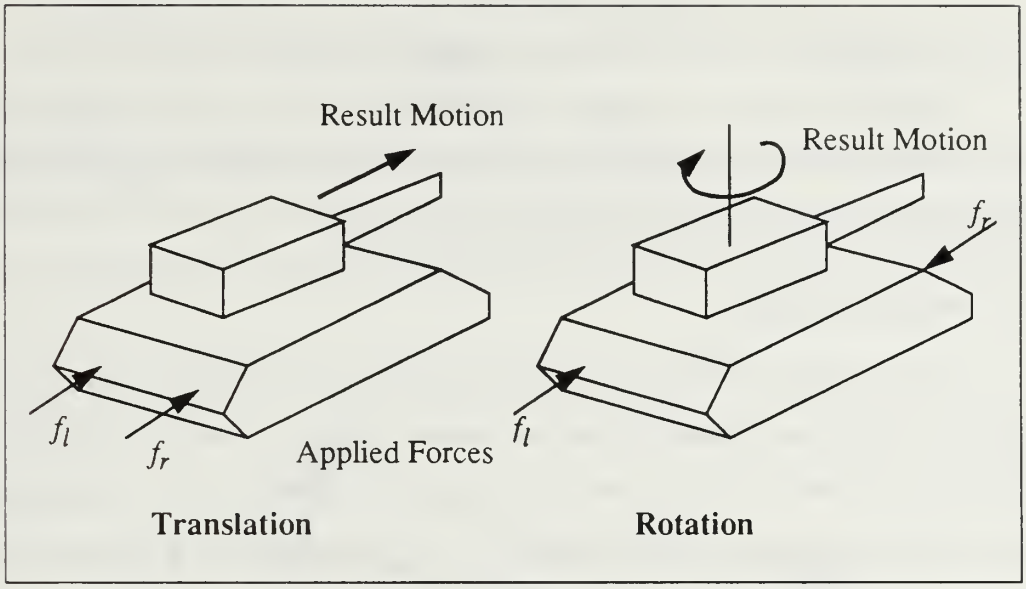


Figure 3.2 Tracked Vehicle Acted On By Two Forces

Tracked vehicles have two internal forces to move the vehicle, the left track force and the right track force. The two tracks of the tracked vehicle are assumed to be parallel. The distances from the center of mass to each track are the same.

The internal forces, f_l and f_r, act on the vehicle at points other than the center of mass and cause vehicle rotation as well as translation. In Figure 3.2, if left and right forces are applied to the vehicle with the same direction and quantity, they produce translational motion without torque. However, two equal opposite forces cause a torque without a linear acceleration. The moments of inertia around the center of mass can be found from Equation 2.12.

The translational factor, velocity, is computed by Equation 2.17 and the vehicle rotation about the center of mass is presented in Equation 2.15. From those equations, each axis' velocities are

$$v_x = v\cos\theta,\ v_z = v\sin\theta \qquad \text{(Eq. 3.1)}$$

The x and z coordinates of new vehicle position can be obtained by Equation 2.19.

Rotational velocity is merely calculated by the sum of two internal forces, $f_t = f_l + f_r$. The opposite direction force has negative value. Torque is computed by this one force and angular orientation is obtained by Equation 2.18.

2. Wheeled Vehicle

The four-wheel vehicles with two fixed axles, such as the M-35 truck, are driven by fixed rear wheels and steered by the front wheels. Since the vehicle is simplified as a block structure, the motion of four wheels are neglected and the direction is computed as a parameter of the front wheels. The moment of inertia of the wheels is neglected and the wheels are making contact with the ground.

The wheeled vehicle model is similar to the tracked vehicle in translational motion. But the steering angle and the vehicle orientation of the wheeled vehicle can be derived by non-holonomic constraints[3] [Shiller et. al., 91]. Choosing a center of mass as the guiding point, and assuming no sliding, makes the vehicle's orientation non-holonomic since only incremental changes for vehicle orientation and steering angle are possible.

The four tires are not treated as individual bodies, but those are considered for the orientation of the vehicle. In Figure 3.1, the r vector is normal to the surface, t is tangent to the path, and q is normal to both. A unit vector, n, in the rq plane is defined in the direction of the center of path curvature. At a velocity, v, the front wheel angle moves the vehicle along the path. In Figure 3.3, d is a distance from front axle to *Center of Mass* and the same for a distance to the rear, since we assume that all vehicles' center of masses are located at the center of vehicles.

---

[3] Non-holonomic constraints mean non-integrable relations involving the coordinates.

The direction of the wheeled vehicle is

$$\tan\alpha = \frac{2d}{\sqrt{\rho^2 - d^2}} \qquad \text{(Eq. 3.2)}$$

where d is half the distance between the axles, and ρ is the distance to the *Center of Turn* from the mass center. The x and z-axis of the vehicle, that are parallel to the terrain, are determined based on the orientation and the steering angle at the previous point [Shiller et. al., 91].

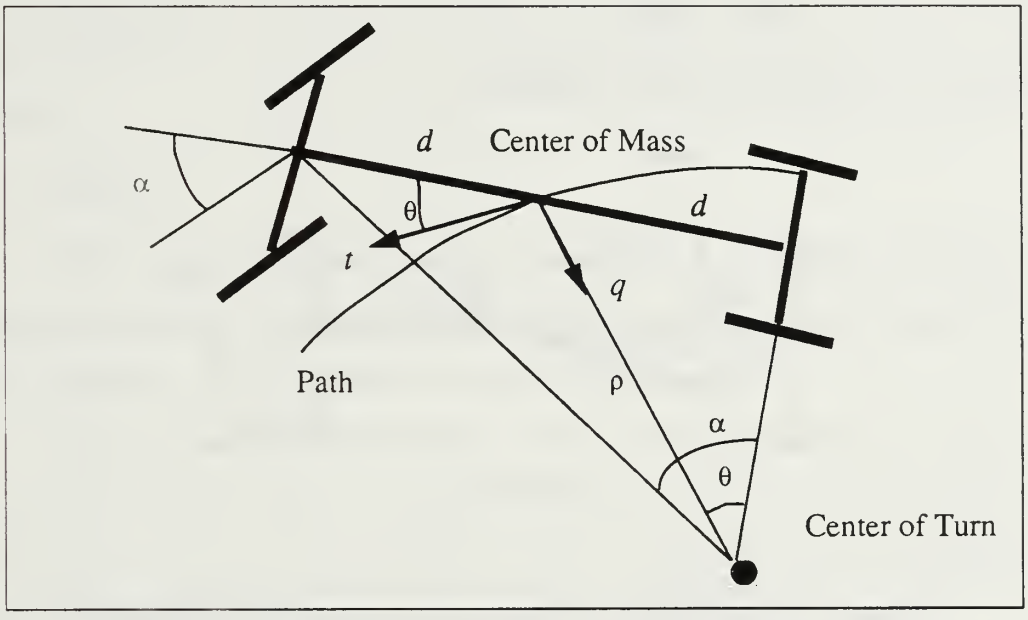


Figure 3.3 Two Degree-Of-Freeedom Wheeled Vehicle Model

The angle θ is a function of the current vehicle position p and the desired direction of motion t. Since p is the result of the previous motions, it can be solved for the incremental changes in θ [Shiller et. al., 91].

Figure 3.4 shows the vehicle at two positions: before and after an incremental move of Δs along the path from some point i. The mass center of vehicle at $i+1$ is located at the end of the vector $\Delta s t_i$ from the mass center of the vehicle at i. During movement, the rear wheel moves along the p_i direction to a distance a from the mass center of vehicle at i. The new location of mass center determines the orientation of the vehicle. Then p_i can be defined as

$$p_i = \cos\theta_i t_i - \sin\theta_i q_i \qquad \text{(Eq. 3.3)}$$

The vehicle's new orientation at $i+1$ is

$$p_{i+1} = \frac{a}{d} p_i + \frac{\Delta s}{d} t_i \qquad \text{(Eq. 3.4)}$$

where steering angle is

$$\alpha = -\Delta s \cos\theta_i + \sqrt{d^2 - \Delta s^2 a \sin\theta_i} \qquad \text{(Eq. 3.5)}$$

Consequently, the angle θ_{i+1} is

$$\theta_{i+1} = \text{acos}\,(p_{i+1} \cdot t_{i+1}) \qquad \text{(Eq. 3.6)}$$

The angle θ_{i+1} is used for the next computation as θ.

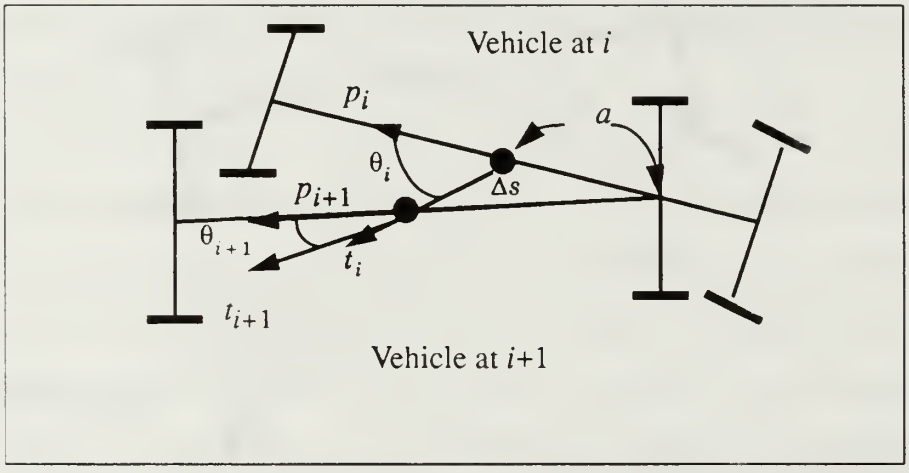


Figure 3.4 Orientation Of Wheeled Vehicle By Two Positions

B. VEHICLE CONSTRAINTS

Several constraints between the vehicle and the environment are considered to ensure vehicle dynamic stability along the path. Various forces such as friction, gravity, engine and brake torque are included in the simulation.

The external forces acting on the vehicle consist of the friction forces between the vehicle and ground, the normal forces and the gravity force. The total external forces are the sum of those forces.

1. Engine And Brake Constraints

Engine torque produces frictional force and pushes the vehicle either forward or backward. A positive engine torque produces a force in the direction of motion, while a negative torque, or a braking force, produces a force in the opposite direction. This force is bounded by the maximum equivalence engine force F_{max} and maximum braking force F_{min}.

$$F_{min} \le f_t \le F_{max} \qquad \text{(Eq. 3.7)}$$

The force f_t is the internal force of the vehicle. The limitation of the forces is validated by the real vehicle specification. The *rpm* value of the engine is changed by internal force not the vehicle speed. The engine torque constraint limits only the acceleration.

2. Friction

Friction is a complicated phenomenon and modeling it exactly is challenging. An approximate treatment is good enough for most engineering purposes, and suffices to produce visually reasonable effects on motion [Wilhelms, 88].

The tracks or wheels are making contact with the ground. At each contact point, there are three unknown forces: two friction and one normal force. Friction forces act to oppose tangential motion along the surface and are dependent upon the normal force pressing into the ground.

Basically, each surface is assigned a stick angle. The intuitive meaning of this angle is that an object will sit motionless on that surface as long as the tilt relative to gravity is less than the stick angle. As soon as the tilt exceeds that limit, the vehicle starts to slide.

The total friction force, F_{fric}, tangent to the xz plane, is represented in path coordinates as

$$F_{fric} = f_t t + f_q q \qquad \text{(Eq. 3.8)}$$

where f_t and f_q are the components tangent and normal to the path, respectively. The equation of motion of the vehicle can be written in terms of the tangential speed v and the tangential acceleration a.

$$f_t t + f_q q - mgk = mknv^2 + mta^2 \qquad \text{(Eq. 3.9)}$$

where k is path curvature, g is gravitational acceleration and m is the vehicle mass. The reaction force in the r direction when the vehicle moves along the bumpy terrain is neglected. The projection of the external forces in the t, q, and r directions is obtained by dot multiplying both sides of Equation 3.6 with the vector t, q and r.

$$f_t = mgk_t + ma \qquad \text{(Eq. 3.10)}$$

$$f_q = mgk_q + mkn_q v^2 \qquad \text{(Eq. 3.11)}$$

In world coordinates, the friction forces in the xz coordinates are

$$F_{fric-x} = f_t \cos\theta - f_q \sin\theta \qquad \text{(Eq. 3.12)}$$

$$F_{fric-z} = f_t \sin\theta + f_q \cos\theta \qquad \text{(Eq. 3.13)}$$

The friction force required by vehicle motion is $\sqrt{F_{fric-x}^2 + F_{fric-z}^2}$, that is divided into dynamic friction and static friction. Dynamic friction is applied to moving vehicles and static friction is applied to unmoving vehicles in the simulation.

3. Gravity

The effect of gravity is easily calculated given the gravitational acceleration (about 9.81 m/sec^2 on the earth's surface). For the center of mass on an inclined surface, the applied gravity force is

$$f_{grv} = (0, -9.81, 0)\, m \qquad \text{(Eq. 3.14)}$$

If the inclination angle is θ, then $\hat{n} = (-\sin\theta, \cos\theta, 0)$. That is the gravity force that applied to the vehicle is

$$F_{grv} = -\hat{n} \cdot f_{grv} = mg\cos\theta \qquad \text{(Eq. 3.15)}$$

If the gravity force F_{grav} exceeds the friction force F_{fric} of the unmoving vehicle on an inclined surface, the vehicle should slide either forward or backward.

The torque due to this force acting in the body fixed coordinate frame is

$$\tau_{grv} = c \times f_{grv} \qquad \text{(Eq. 3.16)}$$

while c is the center of mass and τ is angular torque. This torque acts on the pitching and rolling of the vehicle. The reaction force by this torque is neglected in the simulation. Maximum pitching and rolling angles are determined by the slope of the terrain. Hence during the simulation, vehicles continuously contact the ground. In Figure 3.5, θ_p is pitching angle and θ_r is rolling angle.

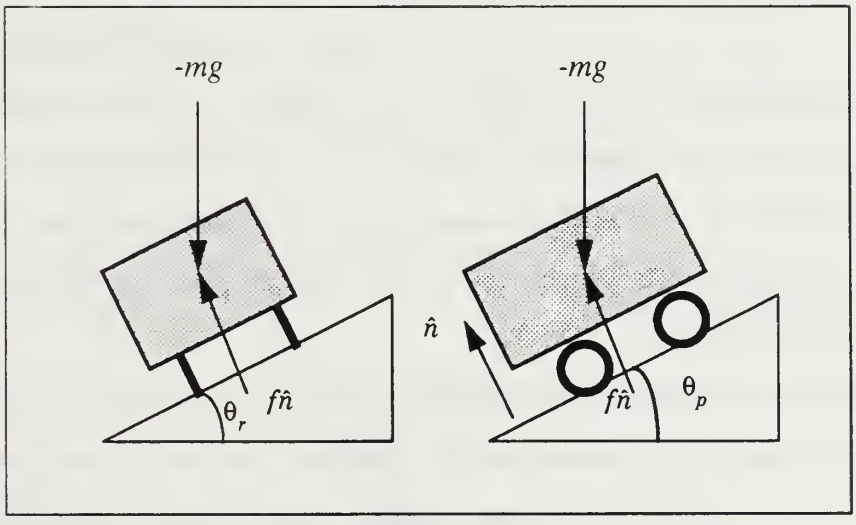


Figure 3.5 Gravity And Mass On An Inclined Surface

IV. BEHAVIORAL CONTROL

A. ADAPTIVE MOTION CONTROL

The problem of motion planning of autonomous vehicles consists of selecting the geometric path and keeping the appropriate vehicle speed so as to avoid obstacles. Adaptive motion control means that the environment has an impact on the vehicle motion and vice versa. Information about the environment and the vehicle such as location and orientation of objects and vehicles must be available during the control process. Adaptive motion makes possible goal-directed and constrained behavior, since it allows the user to describe movement in terms of relations among objects and vehicles. The user only specifies the broad outline of the motion, then the simulation system fills in the details [Zeltzer et. al., 89].

Behavioral control is high-level control, however the motion is achieved by dynamics. It evaluates the state of the environment and generates motion according to certain rules of behavior [Wilhelms, 90]. In the simulation, obstacle avoidance is a basic rule of behavioral motion. Obstacle avoidance is a simple model of adaptive motion control and it is implemented for the autonomous vehicles in the simulation. The goals for behavioral control are to develop effective techniques for planning and following routes based on object information.

In NPSNET, the object locations and terrain elevations are maintained in a text format file. Sensing detects the characteristics of obstacles in the environment and passes this information to the vehicles and the motion control routine. Sensors have a fixed position and orientation on the body. To save memory and improve the frame rate during the execution, the scope of obstacle detection is limited. All fixed objects are attached to the terrain, thus only the geometric data and object data in the field of view are used in obstacle detection.

B. IMPACT OF ENVIRONMENT AND VEHICLE MOTION

1. Searching Range And Detection

Obstacle avoidance motion is defined in two dimensions since all objects are ground-based. The information available on objects is their center coordinates and radius. We develop techniques necessary to detect obstacles with minimum delay between sensing and acting. During the execution, the information concerning the objects is maintained as an array. Only the objects in the field of view are kept in the array. There are more than 40,000 natural and man-made objects in the Fort Hunter-Liggett terrain database used in NPSNET. Distance between an object and a vehicle can be obtained with a two dimensional computation.

The field of view, as the angle in radians, about the present object heading is more than enough. The searching angle is narrow than the field of view. The searching angle θ_l, θ_r is 10 degrees. This angle gives the vehicle with speed 60 km/h approximately 12.9 meters path corridor on the terrain.

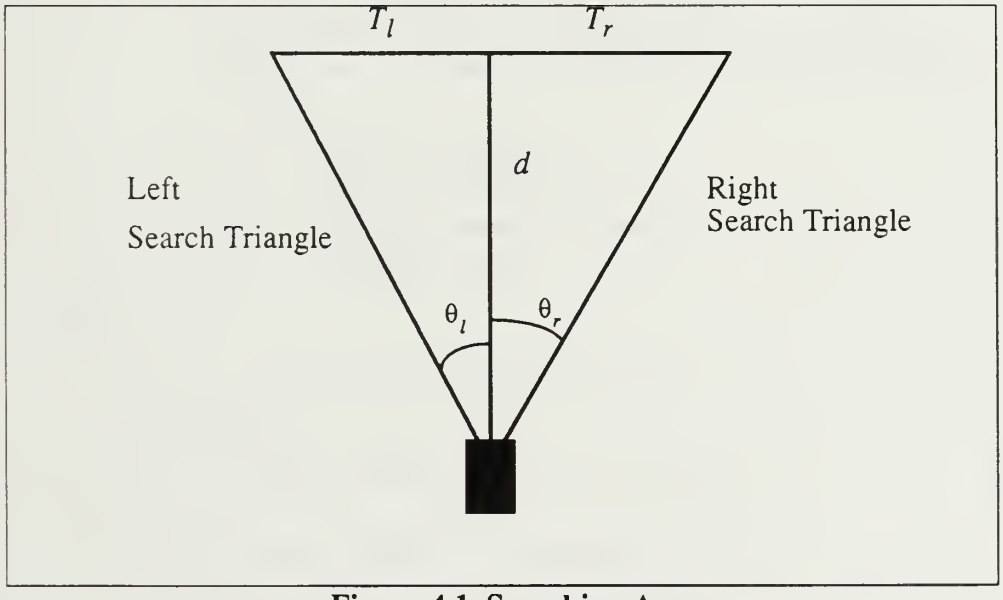


Figure 4.1 Searching Area

All objects are not displayed at one time. The objects which are located in the current field of view, are shown and those are kept in the avoidance object array. The range distance d, is determined by the speed of the vehicle.

$$d = d_{minimum} + d_{speed} \qquad \text{(Eq 4.1)}$$

where $d_{miminum}$ is 20 meters and d_{speed} is the vehicle speed in meters/second.

Our search heuristic is left triangle first. Only one object can be detected at one time. The vehicle can change its path to avoid an object and then the vehicle starts the next search.

```
Determine_Range_Distance ();
Left_search_Angle = Vehicle_Direction - HFOS;
Right_Search_Angle = Vehicle_Direction + HFOS;

/* Sensing coordinates in x,z coordinates */
Sensing_position = Current_Vehicle_Coordinates;

/* Determine left and right search area */
Determine_Boundary_Left_Search_Area ();
Determine_Boundary_Right_Search_Area ();

/* Looking for any obstacles in this area */
Left_Search ();
Right_Search ();

if(LEFT && RIGHT) /* Detect obstacles in left and right area */
        /* If the obstacles are the same one */
        if (Left_Object == Right_Object)
                DETECTION = LEFT;
        else /* If the obstacles are different one */
                /* Measure the distance between the vehicle to the obstacles */
                Left_Distance = Distance_In_2D(Sensing_Position, Left_Object);
                Right_Distance = Distance_In_2D(Sensing_Position, Right_Object);

                if (Left_Distance >= Right_Distance)
                        Detection = RIGHT;
                else /* Left obstacle is nearer than right one */
                        Detection = LEFT;
else
        if (Left)
                Detection = LEFT; /* Obstacle in left area */
        else if (Right)
                Detection = RIGHT; /* Obstacle in right area */
        else
                Detection = FALSE; /* No avoidable obstacle */
```

Figure 4.2 Algorithm For Obstacle Detection

2. Inverse Dynamics

The inverse dynamics problem of a moving vehicle consists of solving for vehicle orientation and translation. Translation and orientation of the vehicle can be derived from dynamics equations presented in Chapters II and III.

The translational inverse dynamics problem is defined as an autocruise control in the simulation. The vehicle velocity is fixed when autocruise mode is selected. To get an appropriate engine force, we need the information of variance of external forces. The engine force must change due to the change of external forces. The summation of new internal forces of the vehicle and total external forces remains constant through the computation. Autocruise speed is computed by this constant force, hence the vehicle velocity is kept fixed. Autocruise speed control helps interactive user control during driving too.

The direction of the wheeled vehicle is presented in Chapter III. Referring to Figure 3.4, the new locations of the mass center and the rear axle define the new vehicle orientation [Shiller et. al., 91]. From Equation 3.4, it is possible to solve the steering angle α that moves the mass center along path in terms of current vehicle's direction θ. The next steering angle α is obtained from the Equation 3.2 and Figure 3.3.

$$\alpha = \text{atan}(2\tan\theta) \qquad \text{(Eq. 4.2)}$$

For the tracked vehicle, total engine force is derived from the autocruise control routine. Then the left and right forces of the vehicle are changed until the vehicle's new orientation matches the desired orientation. Figure 4.3 shows the algorithm for inverse dynamics of the vehicle direction problem.

```
/* For the tracked vehicle */
if (Detect_Object) {
        Compute_Desired_Orientation();
        do while (New_Orientation != Desired_Orientation) {
                if (Object_In_Left_Search_Tri)
                        Left_Track_Force = Left_Track_Force - Delta;
                        Right_Track_Force = Right_Track_Force + Delta;
                else /* Object in right search tri */
                        Left_Track_Force = Left_Track_Force+ Delta;
                        Right_Track_Force = Right_Track_Force- Delta;
                Compute_New_Orientation();
        }
}
/* For the wheeled vehicle */
if (Detect_Object) {
        Compute_Desired_Orientation();
        do while (New_Orientation != Desired_Orientation) {
                Compute_New_Orientation(); /* Equation 4.1 */
        }
}
```

Figure 4.3 Algorithm For Orientation And Desired Path

3. Path Determination

The desired path of the vehicle is simple. The behaviors are grouped into two activities for the set of path determination. The first activity is to travel when the vehicle is in a clear area, and the second is to control the vehicle when obstacles are present. When the vehicle detects an object in the left search triangle, T_l, it changes its direction to the right. The other case, when it detects any object in the right search triangle, it changes its direction to the left. In the case when obstacles are presented in both the left and right triangles, the vehicle avoids the nearest obstacle. The distance is computed two-dimensionally. The new orientation of the vehicle is

$$D_{desired} = D_{current} \pm 0.25\pi \qquad \text{(Eq. 4.3)}$$

where all the values are radians.

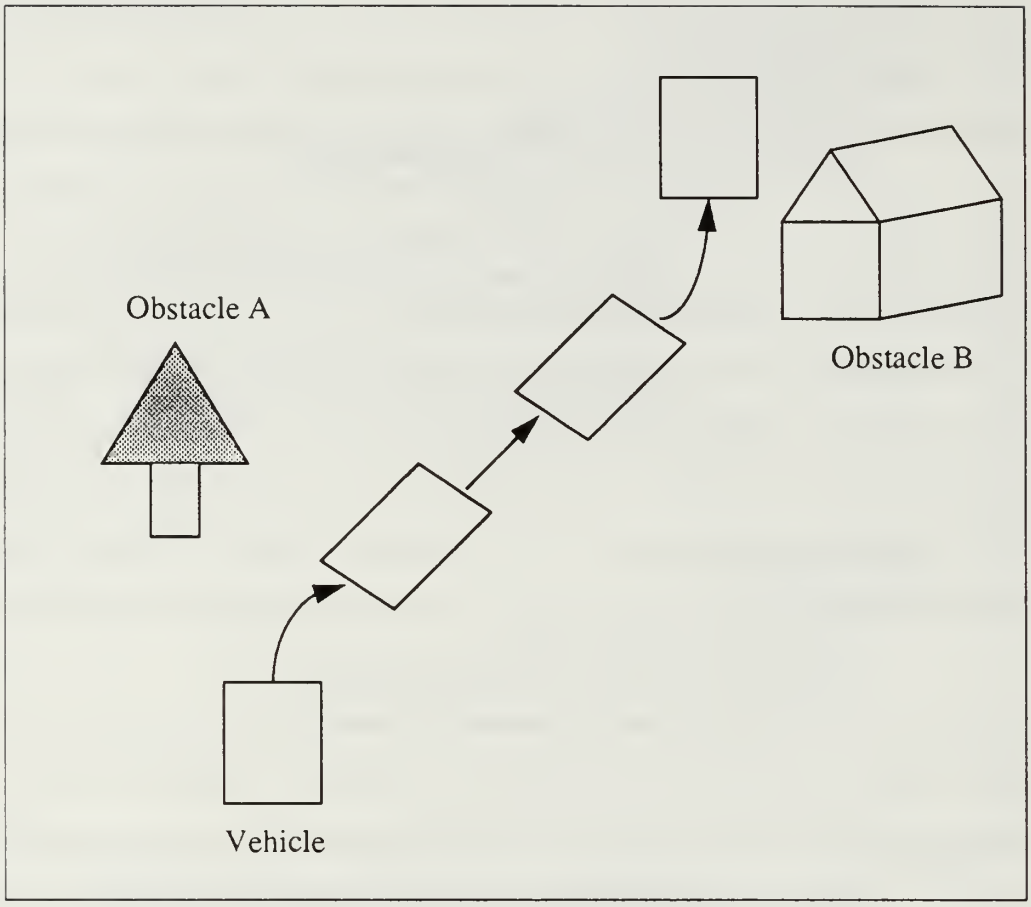


Figure 4.4 Obstacle Avoidance Path

C. FLOW OF BEHAVIORAL MOTION

The behavioral motion control routine is shown in Figure 4.5. First, the vehicle determines the searching distance and checks the object list whether any objects are in the search area or not. When the vehicle detects an object in the left search triangle, it changes its direction to the right. When it detects any objects in right search triangle, it changes its path to the left to avoid the object. The desired direction is matched to the current vehicle orientation, then the vehicle resumes searching again.

Motion is produced by the simplified simulation of dynamics presented in Chapter II and III. Object avoidance control must avoid the nearest imminent obstacle. This method

cause the vehicle to turn and collide with the neighbor obstacle, since we do not make path planning for behavioral motion.

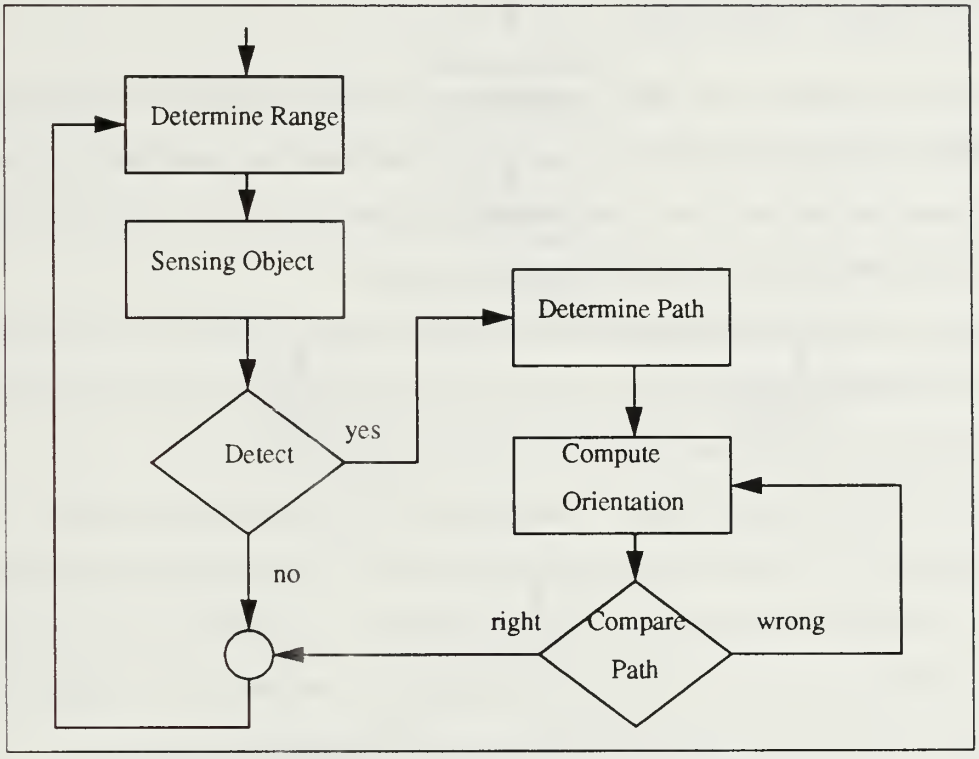


Figure 4.5 Behavioral Motion Control Flow

V. IMPLEMENTATION ISSUES

A. SYSTEM OVERVIEW

Dynamic simulation has been implemented on Silicon Graphics IRIS graphics workstations. The units of the metric system that we use in this work are found in Table 5.1. For that reason, we convert those terms in the simulation then display in the Information Window as user friendly terms such as degrees and km/h.

When the system is initialized, the main window displays the field of view from the vehicle as a default. Vehicle parameters and information are shown in the upper window as text. The flow of control of the system is illustrated in Figure 5.1.

The approach to implementing motion dynamics is pre-processing control. In pre-processing control, two sequential steps are involved. First, appropriate forces and torques are found. All forces and torques are summed to produce a net force and torque acting on a vehicle, and these are used in the second step, dynamic analysis. The advantage of pre-processing is the dynamics control routines can be added and removed easily.

	Display Unit	Computation Unit
Acceleration	kg m /sec^2	kg m /sec^2
Speed	Kilometers / hour	meters /second
Angle	Degree	Radian
Mass	tons	kg
Length	meter	meter

Table 5.1 Parameter Unit Table

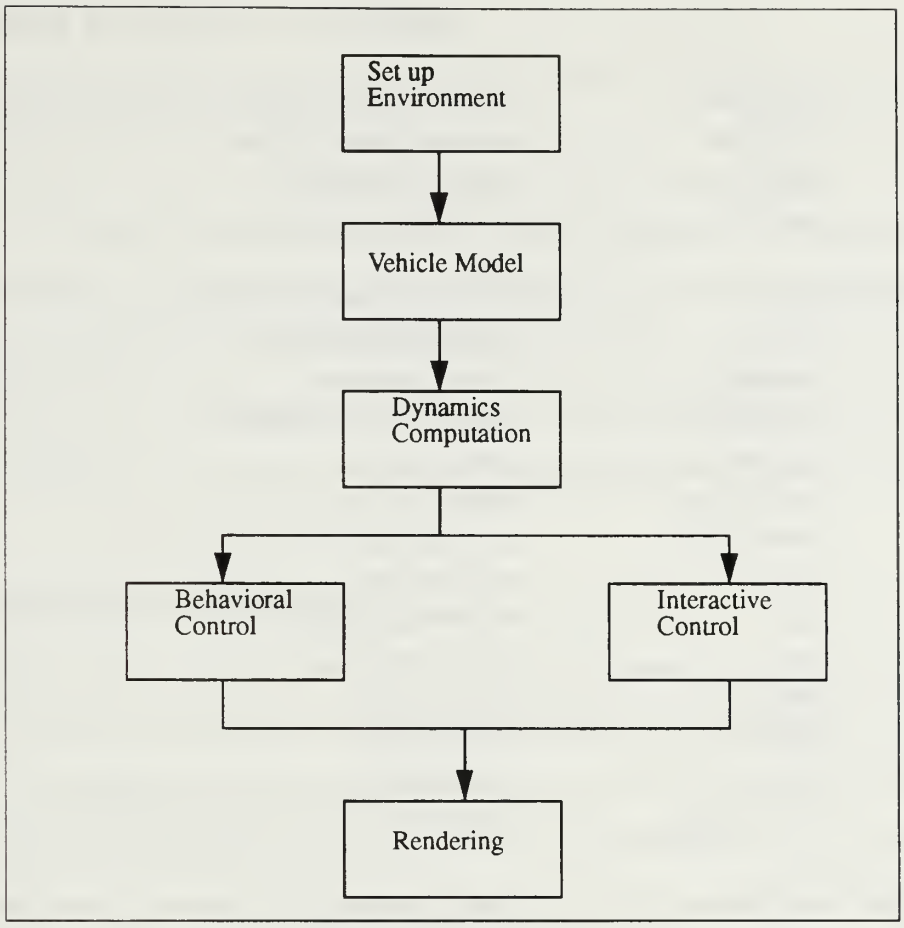


Figure 5.1 System Flow Overview

B. VEHICLE PARAMETERS

The 3D vehicle objects use text based NPSOFF file format [Zyda, 91]. Since NPSOFF is an application independent description of graphical objects, objects can be designed and maintained by general purpose tools [Zyda et.al., 92]. To explore the motion dynamics and behavioral simulation, the vehicle object data structure contains both graphics rendering and motion control variables shown as the following. We use structure type for vehicle parameters shown at Figure 5.2. Figure 5.2 shows variables and their descriptions.

float	limit_speed	maximum vehicle speed
	limit_rpm	maximum engine rpm rate
	max_power	maximum engine power
	min_power	maximum brake power
float	pos[3]	position coordinates in 3-D
	eye[3]	view position coodinates
	lookatpt[3]	viewing coordinates
	lookfmpt[3]	viewing position coordinates
	direction	vehicle direction
	viewdirection	viewing direction
	wheel_angle	front wheel angle for wheeled vehicle
	elev	elevation data
	gas	remained gas amount
	rpm	throttle value
	length	vehicle length
	height	vehicle height
	width	vehicle width
	cruise_speed	cruise speed when the cruise control is on
float	vel[3]	velocity value x, y, z
	scalvel	speed of the vehicle
	acc[3]	acceleration value x, y, z
	scalacc	acceleration of the vehicle
	force	total external force to the vehicle
	lforce	left track force
	rforce	right track force
	surge_force	total engine force of the vehicle
	drag_force	total drag external forces to the vehicle
	g_force	external force by the gravity
	net_torque	net torque to the vehicle
	torque	total torque to the vehicle
	roll	rolling moment
	pitch	pitching moment
	mass	net weight of the vehicle

Figure 5.2 Ground Vehicle Structure

C. VEHICLE MOTION CONTROL

1. Integration Of Forces

The external forces and internal forces are integrated to find new velocity and orientation. All considered forces are defined in Chapter III. Depends on the user selection, the integrated forces are applied to the vehicle's velocity and direction. All applied forces are defined in Chapter III. Figure 5.3 shows the algorithm of the summation of total forces. These forces result the accelerations using first-order equation $f(t, p)$ described in Chapter II. Because of real-time limitations, we use only a single updated evaluation of f in each time step.

2. Vehicle Speed And Direction

The vehicle speed is computed by linear acceleration using Newton-Euler equations. Newton-Euler equation is a first order equation in the translational and angular velocities. To compute the vehicle position and direction, we use improved Newton-Euler equations defined in Chapter II. In kinematically based NPSNET, users can change the vehicle speed using SpaceBall or keyboard directly. And the vehicle speed is kept as a constant unless users do not change the vehicle speed. However in dynamic motion control, the vehicle speed is continuously changed by external forces. This makes the user use dynamic motion control difficult. Figure 5.4 shows the algorithm for vehicle's acceleration and velocity. The maximum velocity is limited by the vehicle specification independently from the variance of forces.

Figure 5.5 shows the algorithm for the computation of vehicle direction. As we mentioned in Chapter III, the vehicle's direction is achieved by two methods, depending on vehicle type. Users can drive the vehicle using either the keyboard or SpaceBall.

The integration of all forces and vehicle speed is shown in Figure 5.6. The new vehicle's linear and angular velocity determine the next position and direction of the vehicle. This parameters render the next frame of the simulation.

```
integration()
{
 /* compute inertia tensor */
 calc_inertia(); /* compute inertia tensor of the vehicle */
 angle_momentum(); /* compute maximum roll and pitch */
 sum_linear_forces(); /* compute surge force and drag force for the vehicle */
 sum_angular_forces(); /* compute vehicle orientation forces */

 if(!behavior) /* user interactive control */
 {
        if(!cruise_flag) /* when cruise mode off */
        {
                /* compute linear and angular velocities */
                calc_acceleration();
                calc_velocity();
                calc_torque();
                calc_direction(,);
        }
        else /* when cruise mode on */
        {
                cruise_on(); /* cruise speed computation */
                velocity = cruise_speed;
                calc_torque();
                calc_direction();
        }
 }
 else /* behavioral control */
 {
        /* compute all the internal and external forces */
        cruise_on();
        velocity = cruise_speed;
        behavior_motion();
        calc_inverse_direction();
        calc_torque();
        calc_direction();
 }
```

Figure 5.3 Algorithm For Integration Of All Forces

```
/* compute the linear acceleration of the vehicle */
calc_acceleration()
{
 /* compute acceleration by the Newton's second law F = ma */
 acceleration = force / mass;

 /* divide the acceleration to x axis and z axis */
 acceleration[X] = acceleration * fcos(vehicle_direction);
 acceleration[Z] = acceleration * fsin(vehicle_direction);
}

/* compute the linear velocity of the vehicle */
calc_velocity()
{
 /* compute the speed from the acceleration in m/sec unit */
 velocity += deltatime * acceleration;

 /* limit the max speed between minimum and the maximum vehicle speed */
 if(velocity <= MINSPEED) velocity = MINSPEED;
 if(velocity >= MAXSPEED) velocity = MAXSPEED;

 velocity[X] = velocity * fcos(vehicle_direction);
 velocity[Z] = velocity * fsin(vehicle_direction);
}
```

Figure 5.4 Algorithm For Vehicle Acceleration And Velocity

```
/* compute angular acceleration and angular velocity by Euler method w = w * dt */
calc_direction()
{

 if(vehicle_type == TRACKED)
 {
        /* omega = torque / inertia tensor */
        rotation = (net_torque / inertia[Y]);
        dt_rot = rotation * deltatime;
        direction += dt_rot;
}

 else /* Wheeled vehicle */
 {
        /* compute the difference between the wheel angle and direction */
        tangle = ftan(wheel_angle);
        rotation = fatan(0.5 * tangle);

        /* if there are difference between the left and right forces
        it should be added to the vehicle direction value */
        if(left_force != right_force)
        {
                delta_rotation = (net_torque / inertia[Y]);
                rotation += delta_rotation;
         }

        /* wheeled vehicle cannot rotate without translation */
        if(velocity != 0.0)
        {
                dt_rot = rotation * deltatime;
                direction += dt_rot;
        }
 }
```

Figure 5.5 Algorithm For Vehicle Direction Computation

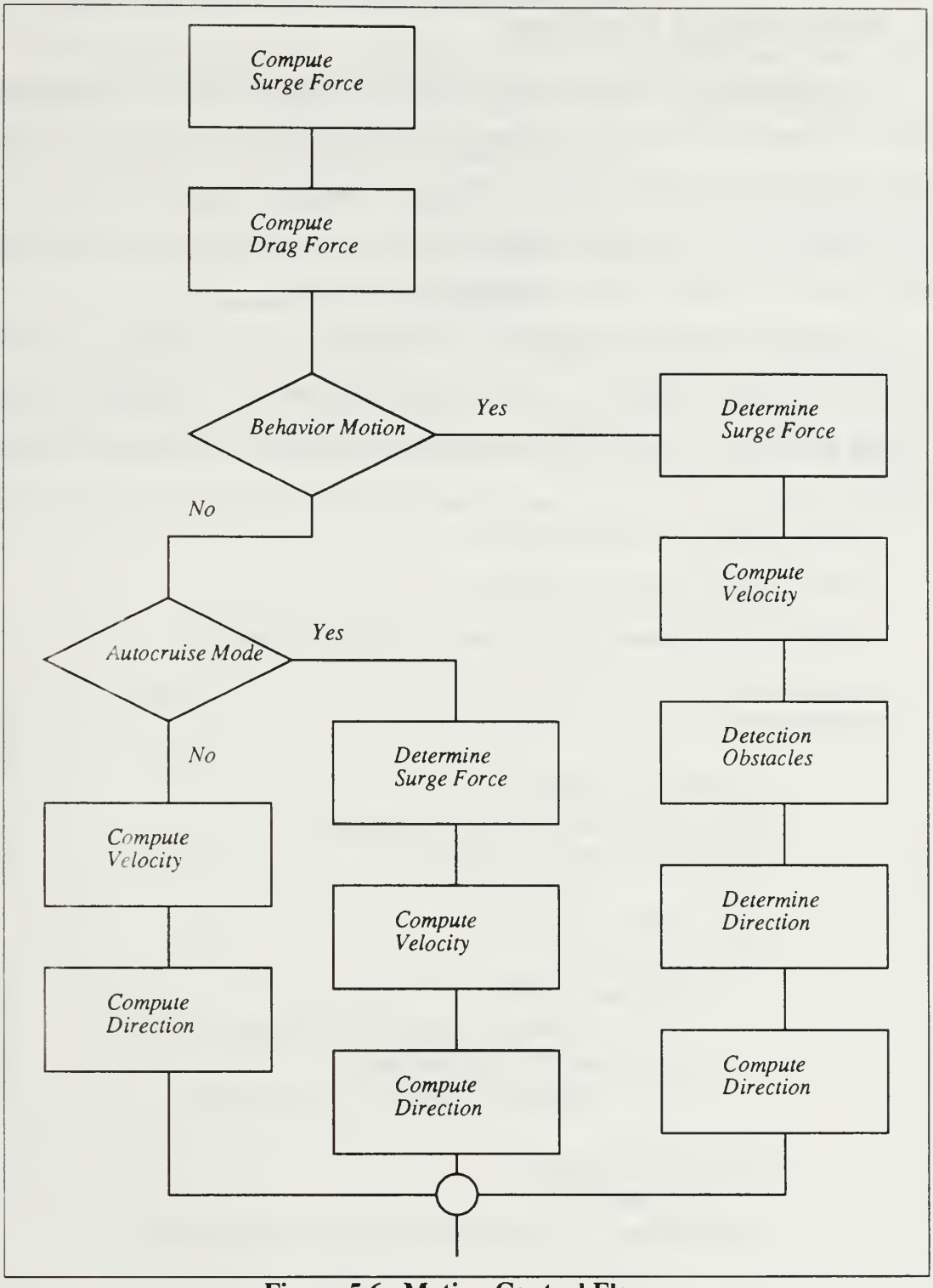


Figure 5.6 Motion Control Flow

D. BEHAVIORAL CONTROL

An autonomous vehicle requires perception, planning and control to act intelligently. Behavioral control is designed to explore use of dynamics both for interactive simulation and automatically controlled simulation. The goal of behavioral control in this simulation is to build a vehicle that can drive autonomously in the cultural and non-cultural objects environment. The behavior rule is avoidance of all obstacles.

The method of obstacle avoidance in this simulation is simple. Figure 5.7 shows the process for obstacle avoidance. A vehicle's behavioral motion is assembled in sensing, planning and control. Function details are attached in Appendix B. The flow of behavioral motion is

Sensing: Obstacle detection, Figure 4.2

Planning: Path determination, Figure 4.3

Control: Inverse dynamics computation for vehicle movement

```
behavior_motion()
{
        search_distance = fabs(velocity * 3.6);
        detect_bounds(); /* determine seraching boundary */
        search_object(); /* obstacle detection */

        if(detect) /* path determination */
        {
                if(whichside == LEFT)
                        dir_value += FINE_PI; /* turn the vehicle right */
                else /* the object is on the right side */
                        dir_value -= FINE_PI; /* turn the vehicle left */
        }
        else /* no avoidable objects */
                dir_value = 0.0;
        inverse_dynamics(); /* compute inverse dynamics for vehicle motion */
}
```

Figure 5.7 Algorithm For Obstacle Avoidance

E. USER INTERFACES AND PERFORMANCE

User interfaces have been developed to show the current status and vehicle information. The upper window in NPSNET displays current status of the system. The Main Window displays the view from the vehicle as a default.

The vehicle dynamics model needs to know what the user is doing. Data acquisition is performed on the steering, brake, and other controls that are available to maneuver the vehicle. User input devices are shown in Appendix A.

This simulation is done in real-time. The performance of NPSNET is not affected by dynamics. Performance of the current simulation system is approximately ten frames per second without texturing, and six frames per second with texturing. We find the effects of dynamic motion control are more realistic than kinematics-based control on bumpy terrain.

IV. CONCLUSIONS AND FURTHER WORK

Current graphics workstations support visual realism and high-performance computing. In this work, we investigate motion accuracy based on the physical laws to improve the NPSNET vehicle simulation. We have used the dynamics of rigid bodies and Euler equations for solving their equations of motion and a behavioral motion based on these dynamics.

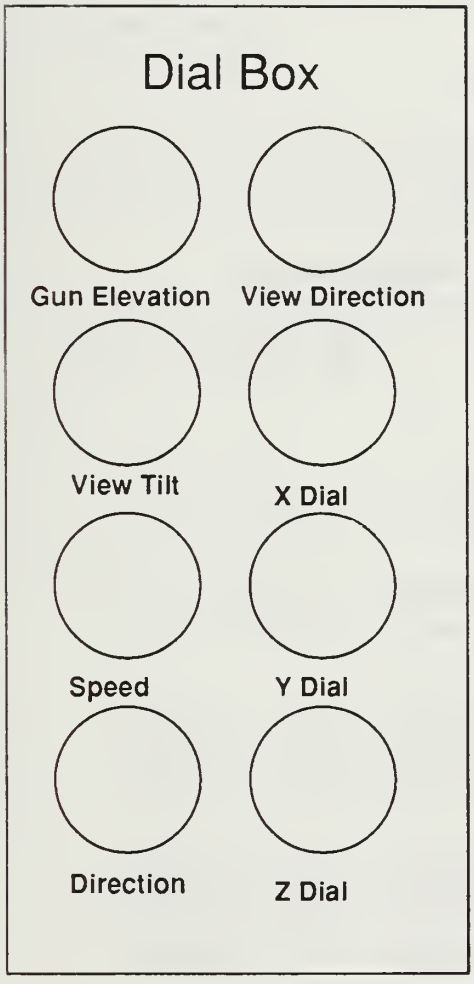
The simulation methods presented in this paper are implemented and run in real-time. Constraint dynamics results in reasonable vehicle motion as we expected. The application of dynamics for high-level motion control is working correctly and we can certify dynamics for low-level motion control in the next extension of the simulation.

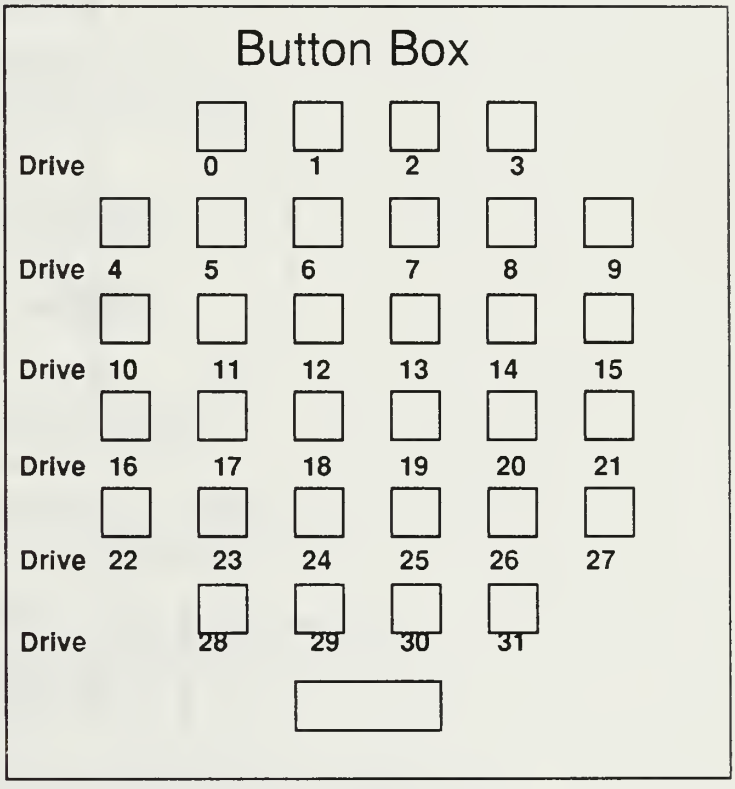
At present, we assume that vehicles are simple block structures to decrease the computational complexity and the Euler equation is selected for the same reason. We have not thoroughly explored all the possibilities for supporting the vehicle model. We do not consider complex natural parameters for this computation such as surface characteristics of the ground, tip-over forces and rebounding forces. For avoidnace motion, since vehicles do not save their past paths, current behavioral motion is not quite intelligent to select optimal path.

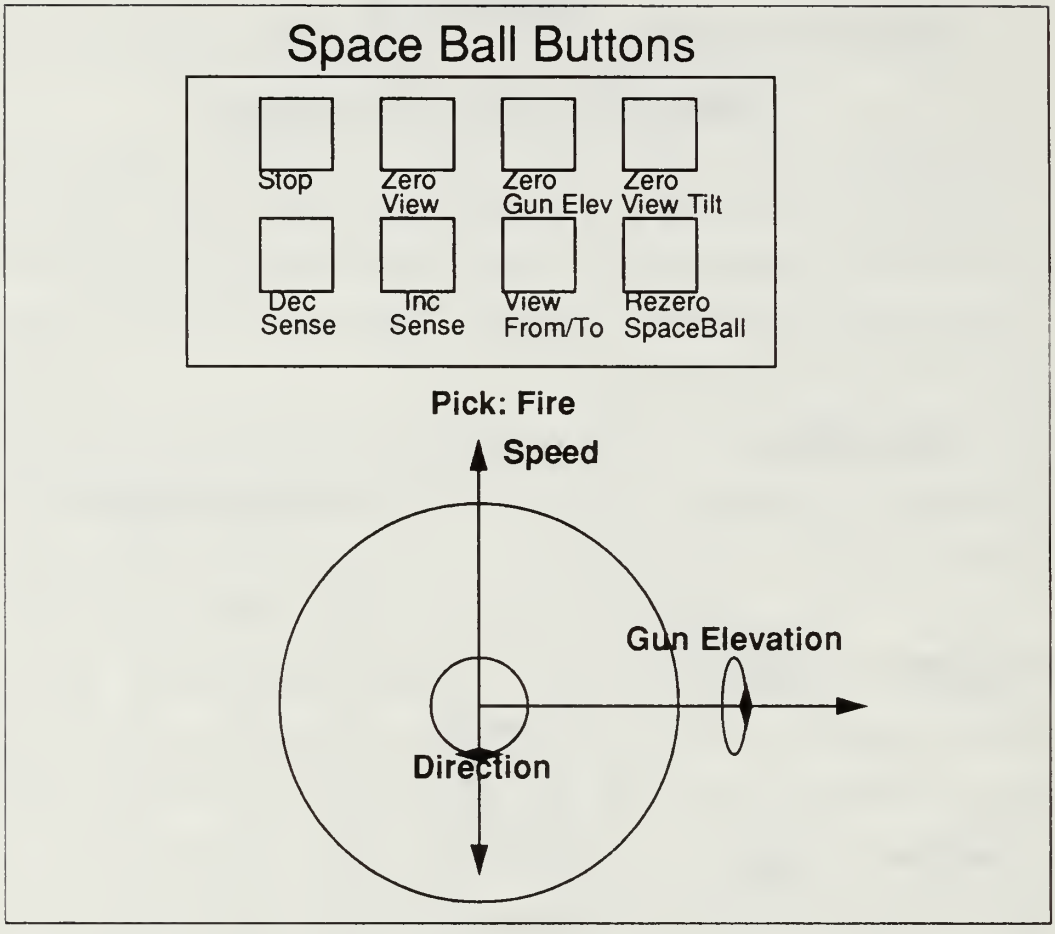
This work suggests a number of topics for further research. Dynamics can be used for the low-level motion control and this can provide for many high-level control issues. The vehicle simulation should handle multi-body vehicles. The higher performance of the graphics workstations and dynamics can lead the vehicle simulation realistic. As hardware performance improves, the vehicle dynamics can take advantage of the faster throughput. And more intelligent behavioral motion is a future avenue of work.

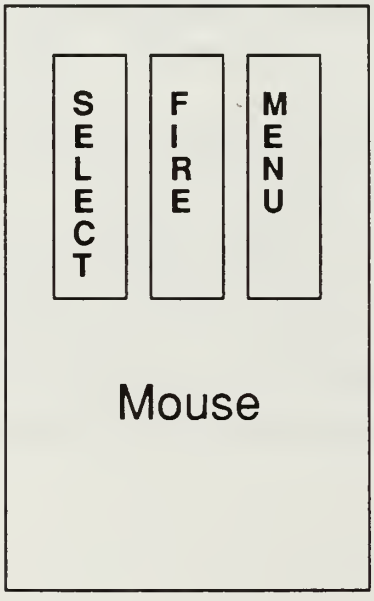
APPENDIX A

USER INPUT DEVICES LAYOUT






Space Ball Buttons
Stop
Zero View
Zero Gun Elev
Zero View Tilt
Dec Sense
Inc Sense
View From/To
Rezero SpaceBall
Pick: Fire
Speed
Gun Elevation
Direction


SELECT
FIRE
MENU
Mouse

APPENDIX B

Source Code For Dynamic Motion Control

```
/*
* Dynamic motion control by the physics laws
* Use Eulerian method for the velocities and positions
*/

#include "vehsim.h"
#include "vehcle.h"

/* External dynamics forces variables in static */
float friction; /* Friction between the vehicle and the terrain */
float air_resist; /* Air resistance to the vehicle */

/* compute the dynamics of the vehicle the tilt and roll */
angle_momentum(vehobj)
struct vehicle *vehobj;
{
    float fp[3],sp[3];
    float xpos,ypos,zpos; /* x,y,z position of center of mass */

    /* The pitch and roll are computed by taking a point 2 meters
    in front to get the pitch and 2 meters to the right sideto get the roll */

    frontpoint[X] = fp[X] = vehobj->pos[X] + fcos(vehobj->direction) * 2.0;
    frontpoint[Z] = fp[Z] = vehobj->pos[Z] + fsin(vehobj->direction) * 2.0;
    frontpoint[Y] = fp[Y] = gnd_level(fp[X],fp[Z]);

    sidepoint[X] = sp[X] = vehobj->pos[X] + fcos(vehobj->direction+HALFPI) * 2.0;
    sidepoint[Z] = sp[Z] = vehobj->pos[Z] + fsin(vehobj->direction+HALFPI) * 2.0;
    sidepoint[Y] = sp[Y] = gnd_level(sp[X],sp[Z]);

    /* get the Y position of the vehicle */
    xpos = vehobj->pos[X];
    zpos = vehobj->pos[Z];
    ypos = gnd_level(xpos,zpos);
```

```
    /* positive pitch is when the nose goes up */
    vehobj->pitch = fatan((fp[Y] - ypos)/2.0) / DEGTORAD;
    if (vehobj->pitch < 0.0) vehobj->pitch += 360.0;

    /* positive roll is when the right side goes down */
    vehobj->roll = fatan((ypos - sp[Y])/2.0) / DEGTORAD;
    if (vehobj->roll < 0.0) vehobj->roll += 360.0;
}

/*
/* compute gravity force on an inclined surface
*/
calc_gforce(vehobj)
struct vehicle *vehobj;

{
    float pitch_angle;
    float rad_angle;

    pitch_angle = vehobj->pitch;

    /* compute the sliding value from the pitching angle of the vehicle */
    if(pitch_angle == 0.0) {
            rad_angle = 0.0;
    }
    /* sliding force is the SIN value of the grade */
    else
    {
            if(pitch_angle > 0.0 && pitch_angle < 90.0) /* the vehicle nose up */
            rad_angle = fsin(pitch_angle * DEGTORAD);

            if(pitch_angle < 360.0 && pitch_angle > 270.0) /* the vehicle nose down*/
            rad_angle = fsin((360.0 - pitch_angle) * DEGTORAD);
    }

    /* f = mg x sinO in case of g=9.81 and ground elevation */
    if(vehobj->scalvel != 0.0)
    {
            if(vehobj->scalvel > 0.0) /* if the vehicle moves forward */
            {
```

```
                if(pitch_angle > 0.0 && pitch_angle < 90.0)
                        vehobj->g_force = (vehobj->mass * GRAVITY * rad_angle);
                else
                        vehobj->g_force = -(vehobj->mass * GRAVITY * rad_angle);
        }
        else /* the vehicle moves backward */
        {
                if(pitch_angle > 0.0 && pitch_angle < 90.0)
                        vehobj->g_force = -(vehobj->mass * GRAVITY * rad_angle);
                else
                        vehobj->g_force = (vehobj->mass * GRAVITY * rad_angle);
        }
    }
    else /* the vehicle speed = 0 */
    {
        if(vehobj->pitch > 0.0 && vehobj->pitch < 90.0)
                vehobj->g_force = -(vehobj->mass * GRAVITY * rad_angle);
        else
                vehobj->g_force = (vehobj->mass * GRAVITY * rad_angle);
    }
} /* end g_force */

/*
/* compute the drag forces to the vehicle, friction specifically
*/
calc_drag_force(vehobj)
struct vehicle *vehobj;
{
    float vel;

    /* compute the air resistance to the vehicle */
    /* approximate the value to the function of square of velocity */
    if(vehobj->scalvel > 0.0)
        air_resist = vehobj->scalvel * vehobj->scalvel * 10;
    else
    {
        if(vehobj->scalvel != 0.0)
                air_resist = -(vehobj->scalvel * vehobj->scalvel * 10);
        else /* the speed is 0 */
                air_resist = 0.0;
    }
```

```
    /* compute the friction between the ground and the vehicle */
    /* approximation coefficient between the velocity and friction */
    vel = fabs(vehobj->scalvel);

    if(!SIDE_BREAK)
    {
        if(vehobj->scalvel == 0.0 && vehobj->scalacc == 0.0)
            /* The vehicle is stalled */
            friction = 0.0;
        else if(vehobj->scalvel == 0.0 && vehobj->scalacc != 0.0)
            /* static friction */
            friction = SFRICTION + 1000.0 + (vehobj->mass);

        else if(vehobj->scalvel > 0.0 && vehobj->scalacc != 0.0)
            /* dynamic friction */
            friction = DFRICTION + (vehobj->mass);

        else if(vehobj->scalvel < 0.0 && vehobj->scalacc != 0.0)
            friction = -(DFRICTION + vehobj->mass);
    }

    vehobj->drag_force = -(friction + air_resist);
} /* emd drag_dorce */

/*
/* compute the engine force to the vehicle and breaking force
*/
calc_engine_force(vehobj)
struct vehicle *vehobj;
{
    if(Stop_Engine == FALSE)
    {
    /* compute the linear translation surge force by the summation of the left and right forces */
        vehobj->surge_force = vehobj->rforce + vehobj->lforce;
        if(vehobj->scalvel > 0.0)
        {
            if(vehobj->surge_force >= MAXPOWER)
                vehobj->surge_force = MAXPOWER;
            if(vehobj->surge_force <= MINBREAK)
                vehobj->surge_force = MINBREAK;
```

```
            }
            if(vehobj->scalvel < 0.0)
            {
                    if(vehobj->surge_force <= MINPOWER) vehobj->surge_force = MINPOWER;
                    if(vehobj->surge_force >= MAXBREAK) vehobj->surge_force = MAXBREAK;
            }

    }
    else
    {
            vehobj->surge_force = 0.0;
            vehobj->torque = 0.0;
    }
} /* end engine force */

/*
/* compute inertia tensor for each axis
*/
calc_inertia(vehobj,inertia,inertia_matrix)
struct vehicle *vehobj;
float inertia[6];
float inertia_matrix[3][3];
{
    /* assume that the vehicle is block wih the same mass density */
    /* principal axes inertia */
    inertia[X] = (vehobj->mass * (vehobj->length * vehobj->height)) / 12.0;
    inertia[Y] = (vehobj->mass * (vehobj->width * vehobj->length)) / 12.0;
    inertia[Z] = (vehobj->mass * (vehobj->width * vehobj->height)) / 12.0;

    /* secondary axes inertia in block structure */
    inertia[XY] = (vehobj->mass * (vehobj->width * vehobj->height)) / 2.0;
    inertia[XZ] = (vehobj->mass * (vehobj->width * vehobj->length)) / 2.0;
    inertia[YZ] = (vehobj->mass * (vehobj->length * vehobj->height)) / 2.0;

    * make an inertia tensor matrix */
    inertia_matrix[X][X] = inertia[X];
    inertia_matrix[X][Y] = -inertia[XY];
    inertia_matrix[X][Z] = -inertia[XZ];
    inertia_matrix[Y][X] = - inertia[XY];
    inertia_matrix[Y][Y] = inertia[Y];
    inertia_matrix[Y][Z] = inertia[YZ];
```

```
    inertia_matrix[Z][X] = -inertia[XZ];
    inertia_matrix[Z][Y] = -inertia[YZ];
    inertia_matrix[Z][Z] = inertia[Z];

} /* end calc_inertia */

/*
* b is the determinant of 3x3 matrix a
*/
det(a,b)
float a[3][3];
float *b;
{
    *b = a[X][X] * (a[Y][Y] * a[Z][Z] - a[Y][Z] * a[Z][Y])
            + a[X][Y] * (a[Y][Z] * a[Z][X] - a[Y][X] * a[Z][Z])
            + a[X][Z] * (a[Y][X] * a[Z][Y] - a[Y][Y] * a[Z][X]);
}

/*
* compute inverse matrixof 3X3 inertia matrix
*/
calc_inverse_inertia(inertia_matrix,inv_inertia_matrix)
float inertia_matrix[3][3];
float inv_inertia_matrix[3][3];
{
    float d;
    det(inertia_matrix,&d);

    inv_inertia_matrix[X][X] = (inertia_matrix[1][1] * inertia_matrix[2][2]
            - inertia_matrix[1][2] * inertia_matrix[2][1])/d;
    inv_inertia_matrix[Y][X] = (inertia_matrix[Y][Z] * inertia_matrix[Z][X]
            - inertia_matrix[Y][X] * inertia_matrix[Z][Z])/d;
    inv_inertia_matrix[Z][X] = (inertia_matrix[Y][X] * inertia_matrix[Z][Y]
            - inertia_matrix[Y][Y] * inertia_matrix[Z][X])/d;
    inv_inertia_matrix[X][Y] = (inertia_matrix[Z][Y] * inertia_matrix[X][Z]
            - inertia_matrix[Z][Z] * inertia_matrix[X][Y])/d;
    inv_inertia_matrix[Y][Y] = (inertia_matrix[Z][Z] * inertia_matrix[X][X]
            - inertia_matrix[Z][X] * inertia_matrix[X][Z])/d;
    inv_inertia_matrix[Z][Y] = (inertia_matrix[Z][X] * inertia_matrix[X][Y]
            - inertia_matrix[Z][Y] * inertia_matrix[X][X])/d;
    inv_inertia_matrix[X][Z] = (inertia_matrix[X][Y] * inertia_matrix[Y][Z]
```

```
            - inertia_matrix[X][Z] * inertia_matrix[Y][Y])/d;
    inv_inertia_matrix[Y][Z] = (inertia_matrix[X][Z] * inertia_matrix[Y][X]
            - inertia_matrix[X][X] * inertia_matrix[Y][Z])/d;
    inv_inertia_matrix[Z][Z] = (inertia_matrix[X][X] * inertia_matrix[Y][Y]
            - inertia_matrix[X][Y] * inertia_matrix[Y][X])/d;
}

/*
* compute torque value for rotation
*/
calc_torque(vehobj)
struct vehicle *vehobj;
{
    if(vehobj->lforce == vehobj->rforce)
            vehobj->net_torque = 0.0;
    else
    {
            if(vehobj->scalvel == 0.0)
            {
                    vehobj->net_torque = vehobj->torque * 0.6;
            }
            else
            {
                    vehobj->net_torque = vehobj->torque * 0.8;
            }
    }
    if(vehobj->type == WHEELED && vehobj->scalvel == 0.0)
    vehobj->net_torque = 0.0;
}

/*
* compute the total linear forces to the vehicle
*/
sum_linear_forces(vehobj, deltatime)
struct vehicle *vehobj;
float deltatime;
{
    struct vehicle tvehpos;

    tvehpos = *vehobj;
```

```
    if(Stop_Engine == FALSE)
    {
            calc_drag_force(&tvehpos); /* compute friction and air resistance */
            calc_gforce(&tvehpos); /* compute gravity force */
            calc_engine_force(&tvehpos); /* compute engine force */
    }

    /* compute the total forces
    each forces are scalar values, so those values can be summed linearly */
    tvehpos.force = tvehpos.drag_force+tvehpos.surge_force+tvehpos.g_force;

    if(tvehpos.force > MAXFORCE) tvehpos.force = MAXFORCE;
    if(tvehpos.force < MINFORCE) tvehpos.force = MINFORCE;

    *vehobj = tvehpos;
}

/*
/* compute the total angular forces to the vehicle
*/
sum_angular_forces(vehobj)
struct vehicle *vehobj;
{
    float abstorque;

    /* compute the angular force to the vehicle by the right hand rule */
    vehobj->torque = vehobj->lforce - vehobj->rforce;

    /* absolute value of torque */
    abstorque = fabs(vehobj->torque);

    /* for the smooth motion for the wheeled vehicle */
    if(vehobj->type == WHEELED && abstorque <= 100.0)
    vehobj->torque = 0.0;

    if(vehobj->torque >= MAXPOWER) vehobj->torque = MAXPOWER;
    if(vehobj->torque <= MINPOWER) vehobj->torque = MINPOWER;
}

/*
* compute the linear acceleration of the vehicle
```

```
*/
calc_acceleration(vehobj)
struct vehicle *vehobj;
{
    /* compute acceleration by the Newton's second law F = ma */
    vehobj->scalacc = vehobj->force / vehobj->mass;

    /* divide the accleration to x axis and z axis */
    vehobj->acc[X] = vehobj->scalacc * fcos(vehobj->direction);
    vehobj->acc[Z] = vehobj->scalacc * fsin(vehobj->direction);
}

/*
* compute the linear velocity of the vehicle
*/
calc_velocity(vehobj,deltatime)
struct vehicle *vehobj;
float deltatime;
{
    /* compute the speed from the acceleration in m/sec unit */
    vehobj->scalvel += deltatime * vehobj->scalacc;

    /* to avoid the numerical instability near 0 speed */
    if(vehobj->scalvel <= 0.3 && vehobj->scalvel >=-0.3)
    vehobj->scalvel = 0.0;

    /* limit the max speed between minimum and the maximum vehicle speed */
    if(vehobj->scalvel <= MINSPEED) vehobj->scalvel = MINSPEED;
    if(vehobj->scalvel >= MAXSPEED) vehobj->scalvel = MAXSPEED;

    vehobj->vel[X] = vehobj->scalvel * fcos(vehobj->direction);
    vehobj->vel[Z] = vehobj->scalvel * fsin(vehobj->direction);
}

/*
* compute angular acceleration and angular velocity by Euler method w = w * dt
*/
calc_direction(vehobj,inertia,deltatime)
struct vehicle *vehobj;
float inertia[6];
float deltatime;
```

```
{
    float rotation; /* rotation value by the torque */
    float dt_rot; /* rotation times deltatime */
    float ext_rot; /* another rotation value for wheeled vehicle */
    float tangle; /* tangent angle for wheeled vehicle */

    if(vehobj->type == TRACKED)
    {
            /* omega = torque / inertia tensor */
            rotation = (vehobj->net_torque / inertia[Y]);
            dt_rot = rotation * deltatime;

            if(dt_rot < 0.0)
                    dt_rot = dt_rot + DPI;
            else if(dt_rot > DPI)
                    dt_rot = dt_rot - DPI;

            vehobj->direction += dt_rot;
            if(vehobj->direction < 0)
                    vehobj->direction += DPI;
            else if (vehobj->direction > DPI)
                    vehobj->direction -= DPI;
    }

    else /* Wheeled vehicle */
    {
    /* compute the difference between the wheel angle and direction */
    tangle = ftan(vehobj->wheel_angle);
    rotation = fatan(0.5 * tangle);

    /* if there are difference between the left and right forces
    it should be added to the vehicle direction value */
    if(vehobj->lforce != vehobj->rforce)
    {
            ext_rot = (vehobj->net_torque / inertia[Y]);
            rotation += ext_rot;
    }

    /* wheeled vehicle cannot rotate without translation */
    if(vehobj->scalvel != 0.0)
    {
```

```
            dt_rot = rotation * deltatime;

            if(dt_rot < 0.0)
                    dt_rot = dt_rot + DPI;
            else if(dt_rot > DPI)
                    dt_rot = dt_rot - DPI;

            vehobj->direction += dt_rot;

            if(vehobj->direction < 0)
                    vehobj->direction += DPI;
            else if (vehobj->direction > DPI)
                    vehobj->direction -= DPI;
    }
    }
}

/*
* compute the wheel angle to direct to the desired direction
*/
calc_inverse_direction(vehobj,des_direction,deltatime)
struct vehicle *vehobj:
float *des_direction;
float deltatime;
{
    float rad_direction;
    float del_angle;

     rad_direction = *des_direction;
    if(vehobj->type == WHEELED)
    {
            if(vehobj->direction != rad_direction)
            {
                    del_angle = fatan(2.0 * ftan(rad_direction));
                    vehobj->wheel_angle = deltatime * del_angle;

                    if(vehobj->wheel_angle >= 45.0)
                            vehobj->wheel_angle = 45.0;
                    if(vehobj->wheel_angle <= -45.0)
                            vehobj->wheel_angle = -45.0;
```

```
            else
            {
                    rad_direction = 0.0;
            }
    }
    else /* tracked vehicle */
    {
            if(vehobj->direction != rad_direction)
            {
                    if(rad_direction > 0.0)
                    {
                            vehobj->lforce += (rad_direction * 10.0) * (vehobj->mass /1000.0);
                            vehobj->rforce -= (rad_direction * 10.0) * (vehobj->mass /1000.0);
                    }
                    else /* desired direction is less than 0 */
                    {
                            vehobj->lforce -= (rad_direction * 10.0) * (vehobj->mass /1000.0);
                            vehobj->rforce += (rad_direction * 10.0) * (vehobj->mass /1000.0);
                    }
            }
            else
                    rad_direction = 0.0;
    }

    *des_direction = rad_direction;
}

/*
* integrate all forces
*/
integration(vehobj,cruise,behavior,deltatime)
struct vehicle *vehobj;
int *cruise;
int *behavior;
float deltatime;
{
    struct vehicle tvehpos;
    float inertia_body[6];
    float path_direction;
    float inertia_matrix[3][3];
    int cruise_flag;
```

```
tvehpos = *vehobj;
path_direction = tvehpos.direction; /* current vehicle direction */
cruise_flag = *cruise;

if(!*behavior)
{
        if(SIDE_BREAK == FALSE)
        {
                /* compute inertia tensor */
                calc_inertia(&tvehpos,inertia_body,inertia_matrix);
                angle_momentum(&tvehpos); /* compute roll and pitch */

                if(!cruise_flag)
                {
                        /* compute all the internal and external forces */
                        sum_linear_forces(&tvehpos,deltatime);
                        sum_angular_forces(&tvehpos);

                        /* compute linear and angular velocities */
                        calc_acceleration(&tvehpos);
                        calc_velocity(&tvehpos,deltatime);
                        calc_torque(&tvehpos);
                        calc_direction(&tvehpos,inertia_body,deltatime);
                }
                else /* cruise on */
                {
                        sum_linear_forces(&tvehpos,deltatime);
                        sum_angular_forces(&tvehpos);
                        cruise_on(&tvehpos);

                        tvehpos.scalvel = tvehpos.cruise_speed;
                        tvehpos.vel[X] = tvehpos.scalvel * fcos(tvehpos.direction);
                        tvehpos.vel[Z] = tvehpos.scalvel * fsin(tvehpos.direction);

                        calc_torque(&tvehpos);
                        calc_direction(&tvehpos,inertia_body,deltatime);
                }
        }
        else /* when the side break is locked */
        {
```

```
                tvehpos.rforce = 0.0;
                tvehpos.lforce = 0.0;
                tvehpos.g_force = 0.0;
                tvehpos.drag_force = 0.0;
                tvehpos.scalvel = 0.0;
            }
    }
    else /* behavioral function */
    {
            /* compute inertia tensor */
            calc_inertia(&tvehpos,inertia_body,inertia_matrix);
            angle_momentum(&tvehpos); /* compute roll and pitch */

             /* compute all the internal and external forces */
            sum_linear_forces(&tvehpos,deltatime);
            sum_angular_forces(&tvehpos);
            cruise_on(&tvehpos);

            tvehpos.scalvel = tvehpos.cruise_speed;
            tvehpos.vel[X] = tvehpos.scalvel * fcos(tvehpos.direction);
            tvehpos.vel[Z] = tvehpos.scalvel * fsin(tvehpos.direction);

            behavior_motion(&tvehpos,&path_direction);
            calc_inverse_direction(&tvehpos,&path_direction,deltatime);
            calc_torque(&tvehpos);
            calc_direction(&tvehpos,inertia_body,deltatime);

            /* inverse dynamics for the vehicle direction */
    }

    /* compute miscelaneous factors of the vehicle */
    calc_gas(&tvehpos,deltatime);
    calc_rpm(&tvehpos);

    *vehobj = tvehpos;
}

/*
* compute the new x and z position by the linear and angular velocity
*/
new_position(vehobj,cruise,behavior,deltatime)
```

```
struct vehicle *vehobj;
int *cruise;
int *behavior;
float deltatime;
{
    struct vehicle tvehpos;
    int behavior_flag;
    int cruise_flag;

    tvehpos = *vehobj;
    behavior_flag = *behavior;
    cruise_flag = *cruise;

    /* compute acceleration and velocity */
    integration(&tvehpos,&cruise_flag,&behavior_flag,deltatime);

    /* Euler Intergration, Pn = Pn-1 + (Vt * dT) + (ACC * dT * dT / 2) */
    tvehpos.pos[X] += (tvehpos.vel[X] * deltatime) + (tvehpos.acc[X] * deltatime * deltatime / 2.0);
    tvehpos.pos[Z] += (tvehpos.vel[Z] * deltatime) + (tvehpos.acc[Z] * deltatime * deltatime / 2.0);
    tvehpos.pos[Y] = gnd_level(tvehpos.pos[X],tvehpos.pos[Z]);

    *vehobj = tvehpos;
}
```

APPENDIX C

Source Code For Behavioral Motion

```
/*
* determine the search tri and find any objects in there
*/

#include "vehsim.h"
#include "vehcle.h"
#include "externs.h"

/* Need a variable to hold the radius of the vehicles' turning that appropriate value to avoid the object */
float RADIUS = 5.0;

int distbetween2(); /* This function finds the distance between 2 points */
int seq_search(); /* search obstacle table array */

/*
* detect non-moving obstacles
*/
detect_nonmoving_objects(vehobj)
struct vehicle *vehobj; /* actual object being checked for avoidance */
{
     float radius_of_vehicle_and_object;
     int xgrid, zgrid; /* gridsquare indices */

     xgrid = (int)(vehobj->pos[X]/GRIDSIZE);
     zgrid = (int)(vehobj->pos[Z]/GRIDSIZE);
 }

/*
*this function constructs the search triangle given the at look at point
*/
detect_bounds(vehobj,vehdir,searchtri,advance)
struct vehicle *vehobj;
float *vehdir;
float searchtri[4][3];
```

```
float advance;

{
    float segdir;
    float langle, rangle;

    segdir = *vehdir;

    /* determine the angles of the view bounds */
    rangle = segdir - HFOS;
    langle = segdir + HFOS;

    if(rangle < 0.0) rangle = DPI + rangle;
    if(langle > DPI) langle = langle - DPI;

    /* compute the center vertex of the triagnle */
    searchtri[2][X] = vehobj->pos[X];
    searchtri[2][Z] = vehobj->pos[Z];

    /* compute the left vertex of the triagnle */
    searchtri[0][X] = searchtri[2][X] + fcos(langle) * (SEARCHOFFSET+advance);
    searchtri[0][Z] = searchtri[2][Z] + fsin(langle) * (SEARCHOFFSET+advance);

    /* compute the right vertex of the triagnle */
    searchtri[1][X] = searchtri[2][X] + fcos(rangle) * (SEARCHOFFSET+advance);
    searchtri[1][Z] = searchtri[2][Z] + fsin(rangle) * (SEARCHOFFSET+advance);

    /* compute the center vertex between the left and right vertices */
    searchtri[3][X] = (searchtri[0][X] + searchtri[1][X]) / 2.0;
    searchtri[3][Z] = (searchtri[0][Z] + searchtri[1][Z]) / 2.0;
}

/*
* serach obstacle array for avoidance
*/
search_object(searchtri,find_object,lrvalue,vehicle_dir)
float searchtri[4][3];
int *find_object;
int *lrvalue;
float vehicle_dir;
{
```

```
float leftminx,rightminx,leftminz,rightminz;
float leftmaxx,rightmaxx,leftmaxz,rightmaxz;
float lobjx, lobjz, robjx, robjz;
int left_search,right_search,overlapped;
int intdir;
float inverse_dir, tempx, tempz;
int ldistance,rdistance;

/* reinitialize the flags */
*find_object = FALSE;
left_search = FALSE;
right_search = FALSE;

boundary(&searchtri[2][X],&searchtri[0][X],&searchtri[3][X],&leftminx,&leftmaxx);
boundary(&searchtri[2][X],&searchtri[1][X],&searchtri[3][X],&rightminx,&rightmaxx);
boundary(&searchtri[2][Z],&searchtri[0][Z],&searchtri[3][Z],&leftminz,&leftmaxz);
boundary(&searchtri[2][Z],&searchtri[1][Z],&searchtri[3][Z],&rightminz,&rightmaxz);

left_search = seq_search(&leftminx,&leftmaxx,&leftminz,&leftmaxz,&lobjx,&lobjz);
right_search = seq_search(&rightminx,&rightmaxx,&rightminz,&rightmaxz,&robjx,&robjz);

/* check if the object is in between the left tri and right tri */
if((lobjx == robjx) && (lobjz == robjz)) {
        overlapped = TRUE;
 }
else {
        overlapped = FALSE;
}

if(vehicle_dir < 0.0)
        vehicle_dir += DPI;

if(vehicle_dir == 0.0 || vehicle_dir == 360.0)
        intdir = 1;
else if(vehicle_dir > 0.0 && vehicle_dir < QTR_PI)
        intdir = 2;
else if(vehicle_dir == QTR_PI)
        intdir = 3;
else if(vehicle_dir > QTR_PI && vehicle_dir < HALFPI)
        intdir = 4;
else if(vehicle_dir == HALFPI)
```

```
        intdir = 5;
else if(vehicle_dir > HALFPI && vehicle_dir < THREE_QTR_PI)
        intdir = 6;
else if(vehicle_dir == THREE_QTR_PI)
        intdir = 7;
else if(vehicle_dir > THREE_QTR_PI && vehicle_dir < DPI)
        intdir = 8;

if(overlapped)
{
        switch(intdir)
        {
                case 1: /* direction = 0 or 2pi */
                        if(searchtri[2][X] >= lobjx)
                                *lrvalue = LEFT;
                        else
                                *lrvalue = RIGHT;
                        break;

                case 2: /* 0 < direction < 0.5pi */
                        inverse_dir = QTR_PI - vehicle_dir;
                        tempx = fabs(lobjx - searchtri[2][X]);
                        tempz = ftan(inverse_dir) * tempx;
                        tempz = searchtri[2][Z] - tempz;
                        if(lobjz >= tempz)
                                *lrvalue = RIGHT;
                        else
                                *lrvalue = LEFT;
                        break;

                case 3: /* direction = 0.5pi*/
                        if(searchtri[2][Z] >= lobjz)
                                *lrvalue = LEFT;
                        else
                                *lrvalue = RIGHT;
                        break;

                case 4: /* 0.5pi < direction < pi */
                        inverse_dir = vehicle_dir;
                        tempz = fabs(lobjz - searchtri[2][Z]);
                        tempx = fabs(ftan(inverse_dir) * tempz);
```

```
            tempx = searchtri[2][X] + tempx;
            if(lobjx >= tempx)
                        *lrvalue = LEFT;
            else
                        *lrvalue = RIGHT;
            break;

case 5: /* direction = pi */
            if(searchtri[2][X] <= lobjx)
                        *lrvalue = LEFT;
            else
                        *lrvalue = RIGHT;
            break;

case 6: /* pi < direction < 1.5pi */
            inverse_dir = vehicle_dir - PI;
            tempz = fabs(lobjz - searchtri[2][Z]);
            tempx = fabs(ftan(inverse_dir) * tempz);
            tempx = searchtri[2][X] - tempx;
            if(lobjx >= tempx)
                        *lrvalue = LEFT;
            else
                        *lrvalue = RIGHT;

            break;

case 7: /* direction = 1.5pi*/
            if(searchtri[2][Z] <= lobjz)
                        *lrvalue = LEFT;
            else
                        *lrvalue = RIGHT;
            break;

case 8: /* 1.5pi < direction <= 2pi */
            inverse_dir = vehicle_dir - THREE_QTR_PI;
            tempx = fabs(lobjx - searchtri[2][X]);
            tempz = fabs(ftan(inverse_dir) * tempx);
            tempz = searchtri[2][Z] - tempz;
            if(lobjz >= tempz)
                        *lrvalue = LEFT;
            else
```

```
                        *lrvalue = RIGHT;
                    break;
            }

            *find_object = TRUE;
}

else /* the object is in left tri or right tri */
{
if(left_search && right_search)
{
            ldistance = dist_in_2d(lobjx, lobjz, searchtri[2][X], searchtri[2][Z]);
            rdistance = dist_in_2d(robjx, robjz, searchtri[2][X], searchtri[2][Z]);

            if(ldistance >= rdistance)
                        *lrvalue = LEFT;
            else
                        *lrvalue = RIGHT;
 }

else /* among right, left or none object */
{
            if(left_search)
            {
                        *find_object = TRUE;
                        *lrvalue = LEFT;
            }
            else
            {
                        if(right_search)
                        {
                                    *find_object = TRUE;
                                    *lrvalue = RIGHT;
                        }

                        else /* !left_search && !right_search */
                        {
                                    *find_object = FALSE;
                        }
            }
}
```

```
} /* the outer else */

}


seq_search(minx,maxx,minz,maxz,objx,objz)
float *minx, *maxx, *minz, *maxz, *objx, *objz;
{
    int ix;
    float tempobjx, tempobjz;

    for(ix=0;ix<=100;ix++)
    {
            tempobjx = avoidlist[ix].center[X];
            tempobjz = avoidlist[ix].center[Z];

            if(tempobjx >= *minx && tempobjx <= *maxx &&
                    tempobjz >= *minz && tempobjz <= *maxz)
            {
                    *objx = tempobjx;
                    *objz = tempobjz;
                    return(TRUE);
            }
    }
    return(FALSE);
}
/*
* determine the minimum and maximum number among the three floating point numbers
*/
boundary(x1,x2,x3,minno,maxno)
float *x1,*x2,*x3;
float *minno,*maxno;
{
    float ix,jx,kx;

    ix = *x1;
    jx = *x2;
    kx = *x3;
    *minno = ix;
    *maxno = ix;
```

```
    if (jx < *minno) *minno = jx;
    if (jx > *maxno) *maxno = jx;
    if (kx < *minno) *minno = kx;
    if (kx > *maxno) *maxno = kx;
}

/*this function computes the 3D distance between two points, it returns an interger*/
dist_in_2d(subjectx,subjectz,objectx,objectz)
float subjectx,subjectz,objectx,objectz;
{
    float distx,distz;
    int dist;

    distx = subjectx - objectx;
    distx = distx * distx;
    distz = subjectz - objectz;
    distz = distz * distz;
    dist = (int)fsqrt((distx+distz));
    return(dist);
}
```

LIST OF REFERENCES

[Badler et.al., 91] Badler, Norman et. al., *Making Them Move*, Morgan Kaufmann Publishing, Inc., San Mateo, CA., 1991.

[Baraff, 91] Baraff, David, "Rigid body concept," *ACM SIGGRAPH Tutorial Note: Physically Based Modeling*, July 1991.

[Cooke, 92] Cooke, Joe, "NPSNET: Flight simulation dynamic modeling using quaternions," *M.S. Thesis*, Naval Postgraduate School, Monterey CA., March 1992.

[Deyo, 89] Deyo, Roderic, "Notes on real-time vehicle simulation," *ACM SIGGRAPH Tutorial Note: Implementing And Interacting With Microworld*, July 1989.

[Issacs et.al., 87] Issacs, Paul and Cohen, Michael, "Controlling dynamic simulation with kinematic constraints, behavior functions and inverse dynamics," *ACM SIGGRAPH Tutorial Note: Computer Animation: 3-D Motion Specification and Control*, July 1987.

[Jurewicz, 89] Jurewicz, T., "A Real Time Autonomous Underwater Vehicle Dynamic Simulator," *M.S. Thesis*, Naval Postgraduate School, Monterey, CA., June 1989

[Meriam et. al., 86]

Meriam, James and Kraige, L.G., *Dynamics 2nd ed.* John Wiley And Sons, New York, N.Y., 1986.

[Shiller et. al., 91] Shiller, Zvi and Gwo, Yu-Rwei, "Dynamic motion planning of autonomous vehicles," *IEEE Transactions on Robotics and Automation*, Vol.7 No.2 April 1991, pp. 241-249.

[Thorpe, 87] Thorpe, Jack A., "The New Technology of Large Scale Simulator Networking: Implications for Mastering the Art of Warfighting," *Proceedings of the Ninth Interservice Industry Training Systems Conference,* November 1987.

[UCF/IST, 90] UCF/IST, "Military Standard (DRAFT) for Protocol Data Units for Distributed Interactive Simulation," University of Central Florida Institute for Simulation and Training, June 1990.

[Wilhelms, 87] Wilhelms, Jane, "Using Dynamic Analysis for Realistic Animation of Articulated Bodies," *IEEE Computer Graphics and Applications*, Vol 7 No 6, June 1987, pp. 12-27.

[Wilhelms, 88] Wilhelms, Jane et. al., "Dynamic animation: interaction and control," *Visual Computer*, Springer-Verlag, 1988, pp. 283-295.

[Wilhelms et. al., 90]

Wilhelms, Jane and Skinner, Robert, "A Notion for Interactive Behavioral Animation Control," *IEEE Computer Graphics and Applications*, May 1990, pp. 14-22.

[Zeltzer et. al., 89]

Zeltzer David, et. al., "An integrated graphical simulation platform," *Proceedings of Graphics Interface 89*, June 1989, London, Ontario, pp. 266-274.

[Zyda, 91] Zyda, Michael J., *Book 7, Computer Graphics*, Naval Postgraduate School, Monterey, CA., April 1991.

[Zyda et. al., 92] Zyda, Michael J., Pratt, David R., Monahan, James G. and Wilson, Kalin P., "NPSNET: Constructing a 3D Virtual World," *Proceedings of the 1992 Symposium on Interactive 3D Graphics*, March 1992.

INITIAL DISTRIBUTION LIST

Defense Technical Information Center Cameron Station Alexandria, VA 229314	2
Dudley Knox Library Code 52 Naval Postgraduate School Monterey, CA 93943	2
Chairman, Code CS, Department of Computer Science Naval Postgraduate School Monterey, CA 93943	1
Dr. Michael J. Zyda Naval Postgraduate School Code CS/Zk, Department of Computer Science Monterey, CA 93943-5100	2
David R. Pratt Naval Postgraduate School Code CS/Pr, Department of Computer Science Monterey, CA 93943-5100	2
Dr. Sehung Kwak Naval Postgraduate School Code CS/Kw, Department of Computer Science Monterey, CA. 93943-5100	2
Captain Hyun Kyoo Park Dongdaemoon-Gu Hoegi-Dong 109-144 #301 Seoul, Korea 130-050	4